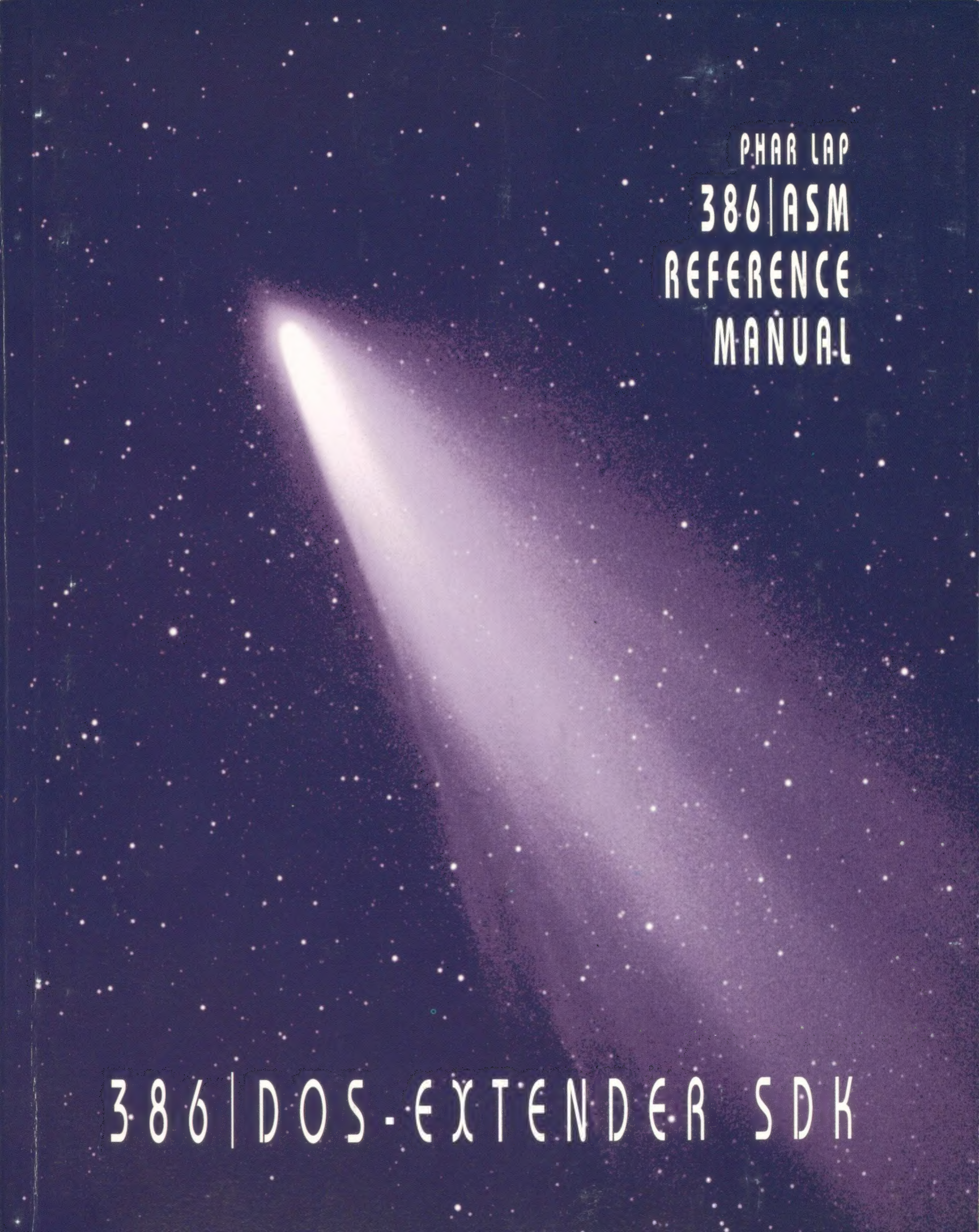
PHAR LAP

386|ASM REFERENCE MANUAL



386|DOS-EXTENDER SDK

386 | ASM
Reference Manual

Phar Lap Software, Inc.
60 Aberdeen Avenue, Cambridge, MA. 02138
(617) 661-1510, FAX (617) 876-2972
dox@pharlap.com
tech-support@pharlap.com

First edition: February 1987.
Second edition: January 1991.

Table of Contents

Tables

Preface

This manual, *386 | ASM Reference Manual,* is intended for experienced systems developers. The programmers who will use it are assumed to be familiar with other, associated Phar Lap products, and to be well-informed on Intel architecture. This manual is not intended as a teaching device; it is for reference and informational purposes only. For other books and manuals on the process, please see the Related Documentation and Books section at the end of this preface.

If, after you read this book, you find that you have suggestions, improvements or comments that can make this a better manual, please call us, write us or send us mail at:

Phar Lap Software, Inc.
60 Aberdeen Ave., Cambridge, MA 02138
(617)661-1510, FAX (617)876-2972
dox@pharlap.com
tech-support@pharlap.com

Your comments are very welcome.

Manual Conventions

This manual relies on certain conventions to convey certain types of information. On the following pages, these are the conventions.

`Courier`	indicates command line, switch syntax and examples; this typeface was chosen for its close resemblance to screen display and to differentiate actual command lines from documentation.
{ }	Items enclosed in braces are optional. The statement will be valid if optional items are left out.

,...	Indicates that the preceeding item may be repeated an arbitrary number of times, with adjacent items separated by commas.

|	Only one of a set of items separated by vertical bars may be used.

Italics	Items in italics must be replaced with a symbol appropriate to a particular statement. The word in italics is generally a descriptive name that identifies a class of symbols that may be used.

For example, a description of a statement to create 10-byte temporary real constant values is shown in the following notation:

```
DT    digits.{digits}{E{+|-}digits},...
```

This indicates that a single real number is entered as a required integer part and a required decimal point, followed by an optional fraction part, and an optional exponent part that consists of a required character E, an optional + or - sign, and required exponent digits. The DT statement itself can be used to create an arbitrary number of real number constants by separating them with commas. Thus, any of the following statements are valid:

```
DT        1.
DT        1.0,1.1,1.2
DT        1.0E2
DT        1.0E+2
DT        1.0E-2
DT        1.E2
DT        0.
DT        0.E0
DT        314.159E-2,0.31459E1
```

All the examples in this manual use upper case characters for assembler directives and expression operators. Lower case is used for all other assembler reserved words and for all user-defined symbols. This convention is for readability only; upper or lower case may be used for all assembler reserved words and for all user-defined symbols.

Related Documentation and Books

Duncan, Ray. *Advanced MSDOS Programming*, 2nd ed. Redmond, WA:

Duncan, Ray, et al. *Extending DOS*, Addison-Wesley Publishing Company, Inc., 1990. ISBN 0-201-55053-9.

Fernandez, Judi N. and Ashley, Ruth. *Assembly Language Programming for the 80386*. McGraw-Hill, 1990. ISBN 0-07-020575-2.

Intel Corporation. *80386 Programmer's Reference Manual*, Order Number 230985, 1986.

Intel Corporation. *80387 Programmer's Reference Manual*, Order Number 2319175, 1987.

Intel Corporation. *80386 System Software Writer's Guide*, Order Number 231499-001, 1987.

Intel Corporation. *Introduction to the 80386*, Order Number 231252, 1986.

Intel Corporation. *The Concrete Representation of 80286 Module Formats*, Order Number 122723-001, 1985.

Turley, James L. *Advanced 80386 Programming Techniques,* Berkeley, CA: Osborne/McGraw-Hill, 1988. ISBN 0-07-881341-5.

This manual is printed on acid-free, recycled paper.

This manual was produced on a Macintosh IIcx, using MS-Word. The examples illustrated in this manual were developed with 386 | ASM, version 3.0.

Chapter 1

Introducing 386 | ASM

386 | ASM is an assembler for the Intel 80386 microprocessor. It is a member of the 80386 Software Development Series from Phar Lap Software, Inc. 386 | ASM is used to assemble one or more assembly language source code files into a single object module. 386 | ASM also assembles programs for the 8086, 8088, 80186, 80188, 80286, 80386, and 80486 microprocessors. Versions of 386 | ASM which run on the IBM PC or PC/AT, VAX/VMS, and a number of different UNIX systems are available.

When 386 | ASM is used to assemble code for the 8086, 8088, 80186, 80188, or 80286, it generates object modules that conform to the Intel 8086 Object Module Format (OMF–86). When code is assembled for the 80386, a simple extension to OMF–86, called Easy OMF–386, is used for the object file format. Appendix K describes the Easy OMF–386 object file format.

386 | ASM works with software products which generate Microsoft MASM compatible assembly language source code, or which process Intel/Microsoft standard Object Module Format (OMF–86) object files. 386 | ASM has been tested with the following products, and no problems have been found:

8086 mode:

Phar Lap 386 | LINK	Version 1.0 or later
Microsoft LINK	Version 3.0 or later
Microsoft LIB	Version 3.0 or later

80286 mode:

Phar Lap 386 | LINK	Version 1.0 or later

80386 mode:

Phar Lap 386 | LINK	Version 1.0 or later

Chapter 2

Using 386 | ASM

2.1 Command Line Syntax

The command line used to run the assembler is the name of the assembler task image (386asm on IBM PC systems), followed by a list of file names and switches. The switches are used to override the default operation of the assembler. By default, 386 | ASM assembles one or more input (source code) files and creates an output object file and an output listing file. When errors occur during assembly, an error message is displayed on the screen and written to the listing file. A switch may be used to redirect error messages displayed on the screen to an error list file.

A file name can be specified on the command line as a complete file name (*filename.extension*), or can be given without an extension, in which case 386 | ASM supplies a default file extension. The assembler assumes the following file name extensions when none are specified:

File Type	Default Extension on IBM PC/ MS-DOS	Default Extension on VAX/VMS and UNIX
input source file	.ASM	.A86
output object file	.OBJ	.086
output listing file	.LST	.LIS
output error list file	.ERR	.ERR

In addition, a complete or partial path may be specified with the file name. If none is given, the current default device and directory are assumed.

Switches begin with a minus sign (-) followed by the name of the switch. No spaces are permitted between the minus sign and the switch name. Any arguments to the switch must immediately follow the switch, with

spaces as separators. Adjacent input file names may be separated by spaces and/or commas. Adjacent switches, or an adjacent switch and input file name, must be separated by spaces. Input file names and switches may be placed in any order on the command line. Input file names may not begin with a minus sign, so that 386|ASM can distinguish between file names and switches. Some examples of valid command lines are shown below, and more examples are given in Appendix L.

Example:

Assemble the input files TEST1.ASM and TEST2.ASM to generate an object file TEST1.OBJ, a listing file TEST1.LST, and an error list file TEST1.ERR:

```
386asm test1 test2 -errorlist test1
386asm test1,test2 -el test1
386asm -el test1 test1 test2
```

Assemble the input files TEST1.ASM in the current directory and test2.A in a subdirectory SUBD of the current directory, and create listing and object files in the top level directories LISTD and OBJD on the current default disk:

IBM PC/MS-DOS

```
386asm test1,subd\test2.a -list \listd\test1 -o \objd\test1
```

VAX/VMS

```
xa386 test1,[.subd]test2.a -l [listd]test1 -o [objd]test1
```

UNIX

```
xa386 test1,subd/test2.a -l /listd/test1 -o /objd/test1
```

2.2 Command Line Switches

Command line switches are used to change the default operation of 386|ASM. By default, 386|ASM will:

- Create an object file with the same name as the first input file and the default file extension shown in section 2.1.
- Create a listing file with the same name as the first input file and the default file extension shown in section 2.1.
- Assume the 80386 as the target processor.

- Will not assemble instructions for an 8087 or 80287 floating point coprocessor.
- Perform a case-insensitive assembly (upper and lower case versions of a symbol are considered identical).

Command line switches begin with a minus sign (-) followed by the name of the switch. There are two forms of each switch name: a long form and a short form. Any argument to the switch must immediately follow the switch name, with a space as a separator. If conflicting switches are given on a command line, the last (rightmost) switch takes precedence.

2.2.1 Object File Switches

The -OBJECT switch is used to give the object file a name other than the default name assigned by the assembler, or to place the object file in a directory other than the current default directory.

Syntax:

```
-OBJECT filename
```

Short Form:

```
-O filename
```

Example:

IBM PC/MS-DOS

```
386asm test -o test.o
386asm test -object t.obj
386asm test -o \objdir\test
386asm test -o \objdir\test.o
```

VAX/VMS

```
xa386 test -o [objdir]test
```

UNIX

```
xa386 test -o /objdir/test
```

The -NOOBJECT switch instructs 386|ASM not to generate an object file.

Syntax:

```
-NOOBJECT
```

Short Form:

```
-NOO
```

Example:

```
386asm test -noobject
386asm test -noo
```

The -NODELETE switch instructs 386|ASM not to delete the .OBJ file, even if there are severe errors in the assembly. By default, the .OBJ file is deleted when severe errors occur.

Syntax:

```
-NODELETE
```

Short Form:

```
-NOD
```

Example:

```
386asm test -nodelete
386asm test -nod
```

2.2.2 Listing File Switches

The -LIST switch is used to give the listing file a name other than the default name assigned by the assembler, or to place the listing file in a directory other than the current default directory.

Syntax:

-LIST *filename*

Short Form:

-L *filename*

Example:

IBM PC/MS-DOS

```
386asm test -list test.l
386asm test -l \listdir\test
```

VAX/VMS

```
xa386 test -l DRA1:[listdir]test
```

UNIX

```
xa386 test -l /listdir/test.l
```

The -NOLIST switch instructs 386|ASM not to generate a listing file.

Syntax:

```
-NOLIST
```

Short Form:

```
-NOL
```

Example:

```
386asm test -nolist
386asm test -nol
```

The -NOSYM switch instructs 386|ASM not to generate a symbol table summary at the end of the source file listing.

Syntax:

```
-NOSYM
```

Short Form:

```
-NOS
```

Example:

```
386asm test -nosym
386asm test -nos
```

2.2.3 Error List File Switch

The -ERRORLIST switch is used to specify a file in which to place error messages. By default, error messages are displayed on the terminal. Error messages are always placed in the listing file (if one is generated) even if an error list file is generated.

Syntax:

```
-ERRORLIST filename
```

Short Form:

```
-EL filename
```

Example:

IBM PC/MS-DOS

```
386asm test -errorlist test
386asm test -el test.e
```

VAX/VMS

```
xa386 test -el t.err
```

UNIX

```
xa386 test -el /listdir/test
```

2.2.4 Include Search Directory Switch

The -INCLUDE switch is used to specify one or more directories to search for files included with the INCLUDE directive (please see section 3.5). By default, only the current default directory is searched for include files. If one or more include search directories are specified on the command line, 386 | ASM attempts to locate include files by searching the specified directories in the order they were given on the command line (left to right) before searching the current default directory.

More than one include search directory can be specified by putting more than one -INCLUDE switch in the command line, or by giving a single -INCLUDE switch more than one argument. If multiple search directories are specified with a single -INCLUDE switch, they must be separated by commas, with no spaces between arguments.

The assembler attempts to locate files by appending the file name given in the INCLUDE statement in the source file to the directory specified with the -INCLUDE switch. Include directories should therefore be specified in the same manner used to specify file paths on the host system, and paths relative to the current directory can be used.

Syntax:

```
-INCLUDE dirname{,dirname...}
```

Short Form:

```
-I dirname{;dirname...}
```

Example:

IBM PC/MS-DOS

```
386asm test -include \includes\
386asm test -i \includes\,isubdir\
386asm test -i ..\includes\
```

VAX/VMS

```
xa386 test -i [includes] -i [.isubdir]
xa386 test -i [-.includes]
```

UNIX

```
xa386 test -i /includes/,isubdir/
```

Please see also: INCLUDE (3.5)

2.2.5 Instruction Set Switches

The instruction set switches are used to enable assembly of instructions for a specific processor (8086, 8088, 80186, 80188, 80286, or 80386) and a specific numeric coprocessor (none, 8087, 80287, or 80387). By default, assembly of instructions for the 80386 processor are enabled, and the numeric coprocessor instructions are disabled (the switches to select those options are therefore redundant, but are provided for consistency).

Specific instruction sets may also be enabled by using the instruction set directives (section 8.2). Use of the directives will override the command

line switches. Note that the 80286 and 80386 instruction sets both contain protected instructions that will only be executed by the processor if it is executing at privilege level zero. The 80286 and 80386 instruction sets may therefore be enabled for non-protected instructions, or for the full (including protected) instruction set.

The –8086 switch enables assembly of 8086 and 8088 instructions.

Syntax:

```
-8086
```

Short Form:

```
-86
```

Example:

```
386asm test -8086
386asm test -86
```

The –80186 switch enables assembly of 80186, 80188, 8086, and 8088 instructions.

Syntax:

```
-80186
```

Short Form:

```
-186
```

Example:

```
386asm test -80186
386asm test -186
```

The -80286 switch enables assembly of 80286 non-protected instructions, and 80186, 80188, 8086, and 8088 instructions.

Syntax:

```
-80286
```

Short Form:

```
-286
```

Example:

```
386asm test -80286
386asm test -286
```

The -80286P switch enables assembly of all 80286, 80186, 80188, 8086, and 8088 instructions, including protected instructions.

Syntax:

```
-80286P
```

Short Form:

```
-286P
```

Example:

```
386asm test -80286P
386asm test -286P
```

The -80386 switch enables assembly of 80386 non-protected instructions, 80286 non-protected instructions, and the 80186, 8086, 80188, and 8088 instructions. This is the default mode of the assembler.

Syntax:

```
-80386
```

Short Form:

```
-386
```

Example:

```
386asm test -80386
386asm test -386
```

The -80386P switch enables assembly of all 80386 instructions, including protected instructions.

Syntax:

```
-80386P
```

Short Form:

```
-386P
```

Example:

```
386asm test -80386P
386asm test -386P
```

The -8087 switch enables assembly of instructions for the 8087 numeric coprocessor.

Syntax:

```
-8087
```

Short Form:

```
-87
```

Example:

```
386asm test -8087
386asm test -87
```

The –80287 switch enables assembly of instructions for the 80287 and 8087 numeric coprocessors.

Syntax:

```
-80287
```

Short Form:

```
-287
```

Example:

```
386asm test -80287
386asm test -287
```

The –80387 switch enables assembly of instructions for the 80387, 80287, and 8087 numeric coprocessors.

Syntax:

```
-80387
```

Short Form:

```
-387
```

Example:

```
386asm test -80387
386asm test -387
```

The –NO87 switch disables assembly of instructions for a numeric coprocessor. This is the default mode of the assembler.

Syntax:

```
-NO87
```

Short Form:

```
-NO87
```

Example:

```
386asm test -no87
386asm test -no87
```

Please see also: Instruction Set Directives (8.2)

2.2.6 Case Sensitivity Switches

By default, 386 | ASM is insensitive to the case of user-defined symbols. For example, the symbols "my_sym", "my_SYM", and "MY_SYM" are all considered identical by 386 | ASM. A command line switch can be used to make 386 | ASM perform case–sensitive assembly. If this option is enabled, the three symbols in the above example are all considered to be different. Whether case sensitivity is enabled or not, all upper and lower case versions of assembler reserved words are considered identical.

The -TWOCASE switch enables case–sensitive assembly of input files. When this switch is used, upper and lower case versions of the same symbol are considered to be different.

Syntax:

```
-TWOCASE
```

Short Form:

```
-TWOC
```

Example:

```
386asm test -twocase
386asm test -twoc
```

The -ONECASE switch disables case–sensitive assembly of input files. This is the default mode; this switch is therefore redundant but is provided for consistency.

Syntax:

```
-ONECASE
```

Short Form:

```
-ONEC
```

Example:

```
386asm test -onecase
386asm test -onec
```

2.2.7 Symbol Definition Switch

The -DEFINE switch is used to define text symbols that can be referenced within the source code being assembled. This feature is useful for conditional assembly of blocks of code. The existence of a symbol can be tested with the IFDEF and IFNDEF directives (please see sections 8.4.3 and 8.4.4), and the value of the symbol can be tested with the IFIDN and IFDIF directives (please see sections 8.4.7 and 8.4.8). Note that text symbols can also be created within the source code with the EQU directive. If no string

is specified when the symbol is defined, it is assigned a value equal to the null string.

Syntax:

```
-DEFINE symbol{=string}
```

Short Form:

```
-D symbol{=string}
```

Example:

```
386asm test -define cross_mode
386asm test -d opsys=xenix
```

Please see also: IFDEF (8.4.3), IFNDEF (8.4.4), IFIDN (8.4.7), IFDIF (8.4.8), EQU (4.2.2)

2.2.8 Extra Error Checking Switch

The –FULLWARN switch is used to turn on extra error checking in 386 | ASM. When extra error checking is enabled, 386 | ASM generates warning errors when the use of a specific segment register is implied in an instruction, but no ASSUME directive has been given assigning the segment register to a segment. The purpose of these errors is to remind the programmer to make sure the segment register points to the correct segment.

Warning errors are also generated when forward references to symbols cause NOP instructions to be generated because the assembler reserves more space for the instruction on pass one of the assembly than is actually needed on pass two. It is usually possible to get rid of the NOPs by giving 386 | ASM more information about the forward referenced symbol with the PTR operator (please see section 9.3.7).

2.3 Listing File Description

The listing file created by 386 | ASM shows (1) source code lines from the input file, (2) the object code generated by the assembler for each source line, and (3) summary tables showing all the user-defined symbols. The

listing file is broken up into pages for readability, with a page header printed at the top of each page.

A single line of the listing file looks like this:

offset object source

where *offset* gives the offset, in hexadecimal, from the beginning of the segment to the generated object code; *object* shows the generated object code in hexadecimal; and *source* shows the source code line as it appears in the input file. For example:

```
00000000                        cseg    SEGMENT word public
00000000   B8 00000001                  mov eax,1
00000005                        cseg    ENDS
```

If an error is detected in a particular source code line, an error message is printed in the listing file immediately following the line in which the error appears. The error message identifies the source code file and the number of the line within the source file which caused the error. It also prints an error number and a message describing the error. For a more complete description of specific error numbers, please see Appendix B.

At the end of the listing file, several symbol tables are printed showing all the user-defined symbols in the program. The sections below describe the symbol tables in the order they appear in the listing file. If there are no symbols in a particular category, the symbol table for that category is not printed.

2.3.1 Page Header

386 | ASM prints out a five-line page header at the top of each page of the listing file. The first line identifies the version number of the assembler and gives the date and time of the assembly. The second line is blank. The third line displays the title, if any, specified with the title directive, and has a section and page number. The section number starts at one and increases incrementally under control of the PAGE directive. The page number is set to one at the beginning of each section and is incremented at every page break. The fourth line of the page header displays the subtitle, if any, specified with the SUBTTL directive. The fifth line is blank.

Example:

```
Phar Lap Macro Assembler Version 3.0 Mon Nov 03 09:21:15 1990
Initialization Code                                      Page 1-5
setup_gdt - init global descriptor table

00000000                    setup_gdt  PROC   near
00000000 55                            push   ebp
00000001 8B EC                         mov    ebp,esp
                                       .
                                       .
                                       .
```

Please see also: TITLE (8.3.1), SUBTTL (8.3.2), PAGE (8.3.3)

2.3.2 Statement Lines

A source code statement is printed on a single line of the listing file in the following format:

```
offset object source
```

Offset gives the offset, in hexadecimal, at which the object code is located within its segment.

Object shows the object code generated for the statement; if no object code is generated, this field is blank. Normally, object code is printed as a string of hexadecimal bytes in the same order in which they are output to the object file, with a space between each byte. However, word and doubleword values such as constants or offsets within a segment are printed, most significant byte first for readability. No spaces are printed between the bytes of word and doubleword values that are ordered most significant byte first. Several special characters are used in the object code field in the listing file. Table 2-1 lists each special character and its use.

The *source* field in the listing file contains the source code statement, copied exactly as it appears in the source code file. One of several special characters may be printed immediately to the left of the source code field. Table 2-2 lists the special characters used with the source code field and their meanings.

TABLE 2-1
SPECIAL CHARACTERS IN OBJECT CODE FIELD

Character	Usage	Description
R	*value* R	The *value* is relocatable; its final value is determined when the program is linked or loaded.
E	*value* E	The *value* is defined externally to the module being assembled; its final value is determined when the program is linked.
----	---- R	A segment selector value. The final value is determined when the program is loaded for execution.
=	= *value*	A constant symbol whose hexadecimal value is *value* has been created.
	= *mnemonic*	An alias symbol for the assembler reserved word *mnemonic* has been created.
	=	A text substitution symbol has been created with the EQU directive.
:	xx:	A segment override prefix byte, with hexadecimal value xx, has been output.
/	xx/	A REP or LOCK prefix byte, with hexadecimal value xx, has been output.
|	66|	An operand size override prefix byte has been output.
	67|	An address size override prefix byte has been output.
[]	*count*[*value*]	The hexadecimal value *value* has been replicated *count* times by the DUP operator.
?	??	One or more bytes of uninitialized memory has been reserved with the ? operand, where two ? characters are printed for each byte of reserved space.

TABLE 2-2
SPECIAL CHARACTERS IN SOURCE CODE FIELD

Character	Meaning
I	The source statement was read from a file that was included with the INCLUDE directive.
n	Macro and repeat block expansion depth (one – nine).
+	Macro and repeat block expansion depth is greater than nine

Example:

```
                                        .LALL
                                        INCLUDE macros.asm
                          I m1          MACRO
                          I             REPT     2
                          I             DB       0
                          I             ENDM
                          I             ENDM

                                        ASSUME   cs:cseg
00000000                    cseg        SEGMENT  word public
use32

                                        EXTRN    cval:abs
 = 0000000A                 cval2       EQU      10
 = 0000000B                 cval3       =        11
 = EBP                      stk_frm     EQU      ebp
 =                          copy        EQU      'Copy'
00000000   0000000A[ 61 62] bvar        DB       10 dup('ab')
00000014   43 6F 70 79                  DB       copy
00000021   ????                         DW       ?
                                        m1
                          1             REPT     2
                          1             DB       0
                          1             ENDM
00000023   00             2             DB       0
00000024   00             2             DB       0

00000025   55                           push     stk_frm
00000026   8B EC                        mov      stk_frm,esp
```

```
00000028    66| B8 000A                    mov     ax,cval2
0000002C    B8 00000000 E                  mov     eax,cval
00000031    8D 05 00000000 R               lea     eax,bvar
00000037    2E: A0 00000000 R              mov     al,bvar
0000003D    66| B8 ---- R                  mov     ax,cseg
00000041    F3/ A4                         rep movsb

00000043                          cseg     ENDS
```

2.3.3 Group and Segment Symbol Table

The group and segment symbol table shows all the groups and segments defined in the source code file. Segments within a group are indented. All segments that are not in a group are listed after all groups. Segment entries in the table also show the use attribute, align type, combine type, and class string for the segment.

Example:

```
            GROUPS AND SEGMENTS
Name                Size        Use      Align     Combine    Class

DGROUP . . .        Group
   CONST . .        0000000A    USE32   DWORD      PUBLIC     DATA
   DATA  . .        00000258    USE32   DWORD      PUBLIC     DATA
CSEG    . . .       00001560    USE32   WORD       PUBLIC     CODE
STACK   . . .       00001000    USE32   DWORD      STACK      STACK
```

Please see also: SEGMENT (3.7), GROUP (3.9)

2.3.4 Structure Symbol Table

The structure symbol table shows all the structure definitions and structure fields defined in the source code file (structure variables are shown in the variable symbol table). Fields within a structure are indented. The size in bytes of each structure is given, and the size in bytes, offset in bytes from the beginning of the structure, and data type are shown for each structure field.

Example:

```
          STRUCTURES
Name                  Size          Offset        Type

PTR386 . . . .        00000006
   SOFFS . . .        00000004      00000000      DWORD
   SELECTOR  .        00000002      00000004      WORD
```

Please see also: Structure Definitions (6.6)

2.3.5 Record Symbol Table

The record symbol table shows all the record definitions and record fields defined in the source code file. (Record variables are shown in the variable symbol table). Fields within a record are indented. The size in bits, the mask value, and the initial value are shown for both the record and record fields. Fields also show the offset in bits of the field within the record.

Example:

```
         RECORDS
Name              Size     Offset     Mask          Initial Value

R1 . . . .        16                  000003FF      0000000A
F1 . . . .        2        8          00000300      00000000
F2 . . . .        8        0          000000FF      0000000A
```

Please see also: RECORD (6.9)

2.3.6 Macro Symbol Table

The macro symbol table shows all the macros defined in the source code file. Each entry shows the number of formal parameters and the number of source code lines in the macro.

Example:

```
        MACROS
Name                #Params     #Lines

dbl_add . . .       2           2
dbl_mult  . .       2           8
```

Please see also: Macro Definition (10.2)

2.3.7 Procedure Symbol Table

The procedure symbol table shows all the procedures defined in the source code file. It also shows all local variables, labels, and constants defined within each procedure. For each procedure, the data type (NEAR or FAR), offset of the procedure within the segment, its scope ("Public", if it has been made available to other modules with the PUBLIC directive, "Prvate", if it has not), and the name of the segment in which it is located are shown. Each entry for a local variable, label, or constant gives the same information described in the sections on the variable and label symbol table and the constant symbol table, with one exception. The scope is always given as "Local", indicating a local symbol which may only be referenced from within the procedure in which it is defined.

Example:

```
        PROCEDURES
Name                Type        Offset      Scope       Segment

init_gdt . . .      N PROC      0000000A    Public      cseg
init_idt . . .      N PROC      00001A10    Public      cseg
init_ldt . . .      N PROC      00000420    Public      cseg
init_paget . .      N PROC      00001038    Public      cseg
        LOCAL VARIABLES AND LABELS
   #1 . . . .       N LABEL     00001060    Local       cseg
   #2 . . . .       N LABEL     00001074    Local       cseg
   #ecode . .       WORD        000018F0    Local       cseg
        LOCAL CONSTANTS
   #arg1 . .        Integer     00000008
   #arg2 . .        Integer     0000000C

gdt_clr  . . .      N PROC      00001F08    Prvate      cseg
```

Please see also: Procedure Blocks (5.3), Global Symbol Definitions (Chapter 7)

2.3.8 Variable and Label Symbol Table

The variable and label symbol table shows all variables and labels defined in the source code file, except those that are local to a procedure block. Each entry shows the data type of the label or variable, its offset in the segment in which it is defined, its scope ("Extern", if it has been declared with the EXTRN directive, "Public", if it has been made available to other modules with the PUBLIC directive, and "Prvate", if it is not a global symbol), and the name of the segment in which it is located.

Example:

```
                VARIABLES AND LABELS
Name              Type        Offset      Scope      Segment

ecode . . .       DWORD       0000000A    Prvate     dseg
err_exit   .      N LABEL     00000256    Extern     cseg
flag1 . . .       Record      0000000C    Prvate     dseg
gdt . . . .       Struc       00000010    Public     dseg
```

Please see also: Variables and Data Definitions (Chapter 6), Global Symbol Definitions (Chapter 7)

2.3.9 Constant Symbol Table

The constant symbol table shows all constant symbols defined in the source code file (with the EQU and = directives), except those local to a procedure block. The type of each constant is given: "Integer" for a constant integer value, "Alias" for an alias to an assembler reserved word, or "Text" for a text substitution symbol. The value of each constant is also given. For integer constants, the value is printed in hexadecimal, or the word "Extern" is printed if the symbol was defined with the EXTRN directive. In addition, if it has been made available to other modules with the PUBLIC directive, it is identified as "Public". For alias constants, the symbol is identified as an alias for an instruction mnemonic, an assembler directive, a register name, an expression operator, or an assembler keyword. The assembler reserved word for which the symbol stands is

also printed. For text substitution symbols, the text string that is substituted for the symbol when it is encountered is shown.

Example:

```
          CONSTANTS
   Name                Type        Value

   cast . . . .        Text        (pword PTR gdtlim)
   flg_msk   . .       Integer     0000003C
   gdt_size . .        Integer     Public 00000050
   stk_frame   .       Alias       Register: EBP
```

Please see also: Defining Constants With the EQU and = Directives (Chapter 4), -DEFINE (2.2.7), Global Symbol Definitions (Chapter 7)

Chapter 3

Program Organization

3.1 Introduction

This chapter defines the basic structure of a program assembled with 386 | ASM. The first section describes the format of a source file. Following sections describe the directives dealing with module naming, entry point specification, file inclusion, program segmentation, and location counter control.

The section describing program segmentation is a reference for how to use the segmentation directives. It is not intended as a description of the segmented architecture of the 80386, nor is it a description of how segments can be used to build a program. For a general overview of the 80386 and its segmented architecture, please see Chapter 11.

3.2 386 | ASM Source File Format

During an assembly, 386 | ASM reads statements from one or more source files and writes output to an object file and a listing file. The object file contains a binary representation of the statements in the source file in a form which can be processed by 386 | LINK. The listing file shows the format of all source statements which were assembled, along with the object code they generated.

A source file is an ASCII text file. It may not contain control codes, except inside string constants, or codes which separate lines of text. A complete description of the valid character set appears in Appendix C.

A source file assembled by 386 | ASM contains zero or more source statements, followed by a source statement containing an END directive.

If the END directive is missing, the assembler supplies one after the last line of the last input file processed. 386|ASM requires all instruction and data bytes emitted to reside in a named section called a segment. A typical source file contains one or more segment definitions before its END statement.

Please see also: Syntactical Elements (Appendix C)

3.2.1 Statement Format

A source code statement must be on a single line in the source file. It can be no longer than 132 characters in length. The format of a source statement is:

```
{identifier} mnemonic {operand,...} {;comment}
```

Identifier is an optional user-defined symbol. *Mnemonic* is an instruction mnemonic or a directive name. This is followed by an optional list of operands to the instruction or directive, separated by commas. An optional comment field begins with a semicolon.

Example:

```
            PAGE
            DW      0,0     ; Allocate two zero words
wvar        DW      1       ; Define a word variable
```

3.2.2 Symbol Formation

User-defined symbols are a sequence of characters from the following set:

```
A-Z
a-z
0-9
_ ? $ @
```

A user-defined symbol may not begin with a digit. Only the first 31 characters in a symbol name are significant; all others are ignored. Upper and lower case versions of a symbol are considered identical unless case sensitivity is enabled with the -TWOCASE command line switch.

Example:

```
sym0       DB         ?
_sym?1     DW         ?
@sym_2     SEGMENT    word public use32
$1:
```

386 | ASM also supports local symbols within procedures. A local symbol is defined by beginning the symbol name with the pound character (#). The rest of the characters in the symbol name follow the normal rules for symbol formation. The pound character may not be used in any character position other than the first character in the symbol name. Local symbols are only known within the scope of the procedure in which they are defined. Thus, a local symbol name may be reused in different procedure blocks without causing multiply defined symbol errors.

Example:

```
add_dbl    PROC     near          ; add two doublewords

#arg1      EQU      8             ; 1st argument
#arg2      EQU      12            ; 2nd argument

           push     ebp           ; set up
           mov      ebp,esp       ; stack frame

           mov      eax,#arg1[ebp]   ; add two
           add      eax,#arg2[ebp]   ; arguments
           jo       #error        ; set error if overflow
#exit:
           pop      ebp           ; restore original EBP
           ret
#error:    call     ovf_err       ; call error routine
           jmp      #exit

add_dbl    ENDP
```

Please see also: -TWOCASE (2.2.6), Procedure Blocks (5.3)

3.2.3 Constants

386 | ASM supports all of the data types and formats supported by the 8086/8088/80186/80188/80286/80386 and by the 8087/80287 coprocessors.

Integers are encoded in two's complement, and may be one, two, four, eight, or ten bytes in length. Real numbers are encoded in the 80287 floating point format and may be four, eight, or ten bytes in length. Packed decimal numbers are encoded in the 80287 packed decimal format and are always ten bytes in length. Appendix G gives the range of values that can be represented by each data type. All numbers are stored least significant byte first in memory. For a complete description of 80386 and 8087/80287 data formats, please see the list of related documents in the Preface.

Please see also: Data Declaration Directives (6.2), Data Types (Appendix G)

Integer Constants

Syntax:

```
digits{radix_specifier}
```

Integer numbers are given as a string of one or more *digits* followed by an optional one-character radix specifier. *Digits* must be characters in the set 0-9, and A-F or a-f for hexadecimal numbers. If no *radix_specifier* is given, the number is assumed to be in the current default radix (any base from base 2 to base 16). The default radix is initially base 10 and may be changed using the .RADIX directive. Radix specifiers are shown in Table 3-1.

TABLE 3-1
RADIX SPECIFIERS

Radix Specifier	Radix
B or b	binary (base 2)
Q or q	octal (base 8)
O or o	octal (base 8)
D or d	decimal (base 10)
H or h	hexadecimal (base 16)

All numbers, regardless of radix, must begin with a decimal digit (0-9); anything beginning with a letter is treated as a symbol instead of a number. A leading 0 can always be used, if necessary. The examples below all specify the decimal number 12, given in different radixes.

Example:

```
DW      12      ; decimal
DW      0Ch     ; hexadecimal
.RADIX  16      ; set default to hexadecimal
DB      0C      ; hexadecimal 12
DD      14Q     ; octal
DQ      1100B   ; binary
DT      12d     ; decimal
.RADIX  11      ; set default to base 11
DW      11      ; base 11
DW      12D     ; decimal
```

Integer numbers are normally stored in a two's complement format, which may be processed by either the 80386 or the 80287 coprocessor. The DT directive may also be used to create packed decimal numbers, stored in the 8087/80287 packed decimal format. A packed decimal number is created by the DT directive when the default radix is decimal and no radix specifier is used at the end of the number; otherwise, a normal two's complement integer is created. The maximum size of a packed decimal number is 18 digits.

Example:

```
.RADIX  10
DT      12345     ; packed decimal
DT      12345D    ; two's complement integer
DT      12345Q    ; two's complement integer
DT      0ABCDH    ; two's complement integer
```

Please see also: .RADIX (8.6.2)

Real Constants

Syntax:

digits.{*digits*}{E{+|-}*digits*}

Real numbers are given as a required integer part, a required decimal point, an optional fractional part, and an optional exponent. The exponent, if present, begins with the required character "E", and is followed by an optional + or - sign and required digits. All *digits* must be decimal (0-9). Real numbers may only be used with the DD, DQ, and DT data declaration directives. Real numbers are stored in 8087/80287 floating point format.

Example:

```
DD      1.0
DQ      1.0E20
DT      1.
DD      0.002E-10
DD      1.0E+20
```

Encoded Reals

Syntax:

*digits*R

Real numbers may also be entered in an encoded form. The programmer encodes the number in the floating point format used by the 8087/80287 numeric coprocessor (or by some other numeric coprocessor or software floating point package). Encoded real numbers are given as a string of hexadecimal *digits* followed by the real number specifier R. Encoded real numbers may only be used with the DD, DQ, and DT directives, and the number of digits must be exactly the size of the data type. If the number has a leading 0, the number of digits may be, but does not have to be, increased by one. Thus, eight digits must be given with the DD directive, 16 digits with DQ, and 20 digits with DT (or 9, 17, and 21, respectively, with a leading 0).

Example:

```
DD      1.0E1
DD      41200000R

DD      -0.000123
DD      0B900F98FR
```

```
DQ      1.0E38
DQ      47D2CED32A16A1B1r

DT      1.0E1
DT      4002A0000000000000R
```

String Constants

Syntax:

```
"string"
'string'
```

String constants are a sequence of one or more characters enclosed in double quotation marks (") or right single quotation marks ('). The characters in *string* may be any printable ASCII character, any non-printable ASCII character except the NUL (0) character, and any non-ASCII character (a character with the most significant bit set). This allows graphics characters to be directly encoded in character strings.

To encode the quotation mark that encloses the string as part of the string, the quotation mark character must be typed twice.

String constants may be used with the DB directive to enter the character string in memory. Strings up to eight characters in length may also be used as integer values, provided they will not overflow the specified data type. The ASCII value of each character in the string is stored as one byte in the integer, with the rightmost character in the string stored as the least significant byte in the integer.

Example:

```
DB      'this is a string'
DB      "don't"
DB      'don''t'
DB      1000 DUP ("STACK")

; string encoded integers

DW      'ab'
DD      'abcd'
DQ      'abcdefgh'
DW      'ab'DUP(?)
```

3.3 NAME Directive

Syntax:

```
NAME modname
```

The object file output by the assembler contains a header record with the name of the assembled module. 386|LINK uses this name when the module needs to be referenced.

The NAME directive is used to set this module name in the object output file. The *modname* parameter can be a string of any length, though only the first 132 characters are significant. Only one NAME directive is permitted per module being assembled. If a module contains more than one NAME directive, an error is posted for each attempt to change the name, and the name specified by the first NAME directive is used.

If the name of the module being assembled is not specified, the assembler assigns a default name using the following logic:

1. If a title for the listing file pages is specified with the TITLE directive, the first six characters of the title string (or less if the string is not long enough) are used as the module's name.

2. If the module has no name and a title is not specified, the assembler assigns the name of the source code file to the module.

Example:

The following example creates a module named mod1:

```
TITLE This is the first module of the system
NAME mod1
END
```

If no name is specified, the first characters of the title are used, if available. This example creates a module named MTH386:

```
TITLE MTH386 - An 80386 long arithmetic library
END
```

Please see also: TITLE (8.3.1)

3.4 END Directive

Syntax:

```
END {entry_pt}
```

The END directive is used to terminate assembly of the current module and to specify an optional entry point into the program. When the assembler encounters an END directive, it treats it as if it reached the end of the input file. Any text following the directive is ignored and is not copied into the listing file. If the assembler encounters an END statement while processing a file included with the INCLUDE directive, the effect is the same. All input files are closed, and the assembly is terminated.

The optional entry point specifier *entry_pt* specifies the value to be loaded into the 80386 instruction pointer when execution begins. It can be any expression which evaluates to an address in the module. Only one module in a program may specify an entry point; otherwise, an error occurs when the program is linked.

Example:

The END directive in the example below terminates the assembly of the module and establishes the procedure *start* as the entry point of the program:

```
TITLE Example of entry point specification with END

code   SEGMENT word public

start  PROC far

       mov eax,1       ; This will be the first
                     ; instruction

start  ENDP

code   ENDS

END    start
```

3.5 INCLUDE Directive

Syntax:

```
INCLUDE filename
```

The INCLUDE directive is used to insert text contained in another file into the file being assembled. A file which is included typically contains definitions for a system's data structures, global variables, and public procedures. Include files may be nested to any depth.

The *filename* argument specifies the name of the file to be included in the assembly. The assembler first searches any include library directories specified on the command line by appending the *filename* parameter to each directory in its list and attempting to open the file using that name. If this fails, it attempts to open the file using the name specified.

Example:

The following example program includes definitions for its data structures and external functions from header files:

```
TITLE Example of using INCLUDE

INCLUDE dstruct.h
INCLUDE extern.h

code    SEGMENT word public

start   PROC far

        mov  ax,field1[ebx]     ; field1 is defined in
                                ; dstruct.h
        call print_field        ; print_field is defined in
                                ; extern.h
        ret

start   ENDP

code    ENDS
END
```

Please see also: .LISTI (8.3.6), .XLISTI (8.3.7), -INCLUDE (2.2.4)

3.6 COMMENT Directive

Syntax:

```
COMMENT delim text delim
```

The COMMENT directive causes the assembler to treat the *text* between the two occurrences of the delimiter *delim* as a comment. Comments are copied to the listing file but are not processed by the assembler. The first non-whitespace character past the directive is the delimiter for the comment. The second occurrence of the delimiter is not required to be on the same line as the COMMENT directive, thus permitting multi-line comments. The remainder of the text on the same line as the second delimiter is also treated as a comment.

Example:

An example of a multi-line comment follows:

```
TITLE An example of a multi-line comment
COMMENT *

The text contained on these lines will be treated as a
comment because the second occurrence of the delimiter does
not occur until below.

* End of COMMENT
END
```

3.7 SEGMENT Directive

Syntax:

```
segname SEGMENT {align} {combine} {useatr} {access}
{'class'}
```

The SEGMENT directive marks the beginning of a logical section of the module being assembled. All instruction and data bytes generated by the assembler are placed in the segment named *segname* until the ENDS directive is processed for that segment. When program modules are combined by the linker, all segments with the same name are concatenated and relocated.

The *segname* parameter specifies the name of the segment being opened. The optional parameters following the directive provide information for the linker and loader as to how the segment is to be used. These optional parameters may appear in any order, and the assembler decides which type each parameter is based on its value. If a parameter is not specified, a default is chosen by 386 | ASM. The logic used to pick a default depends on the parameter and is formally specified in the sections describing each parameter.

386 | ASM permits a segment to be opened and closed more than once during an assembly and treats all segments with the same name as a single segment. When a segment is reopened, its location counter starts at the value it had when the segment was last closed. The assembler, however, requires that any optional segment attribute specifiers for a segment being reopened match those specified the first time the segment was opened.

Segment definitions may be nested. When a segment is opened inside another segment, assembly into the enclosing segment is temporarily suspended. All data emitted by statements inside the nested segment are placed in that segment, and the location counter for the enclosing segment is not modified. When the ENDS directive is processed for the nested segment, assembly continues into the enclosing segment. Though nesting of segment definitions is permitted, these definitions may not overlap. The ENDS directive for a nested segment must precede the ENDS directive for the enclosing segment; otherwise, 386 | ASM posts an error.

The following sections describe each of the optional segment attribute specifiers, the values they may take on, and the significance of each value.

Please see also: ENDS (3.8), GROUP (3.9), ASSUME (3.10)

3.7.1 Align Type Specifier

A segment's align type informs the linker what values are permissible for the base address of the segment. When 386 | LINK combines object modules to create an executable image, it guarantees that each portion of a segment has a base address compatible with its alignment type by padding

the space between segments with null bytes if necessary. The align type specifier can take on any one of the following values:

Align Type	Meaning
BYTE	Segment can be based at any address.
WORD	Segment must be on a word boundary (base address must be even).
DWORD	Segment must be on a double word boundary (base address must be divisible by 4).
PARA	Segment must be on a paragraph boundary (base address must be divisible by 16).
PAGE	Segment must be on a page boundary (base address must be divisible by 256).
PAGE4K	Segment must be on a 4 kilobyte page boundary (base address must be divisible by 4096).

If no align type is specified for the segment, a default of PARA is used.

3.7.2 Combine Type Specifier

A segment's combine type tells the linker how to combine segments having the same name. The combine type specifier must have one of the following values:

Combine Type	Meaning
PUBLIC	Segments with the same name having a PUBLIC combine type are concatenated by the linker to form a single segment. Each portion of the segment is relocated relative to the new segment base as it is combined.

STACK	Segments with the same name having a STACK combine type are concatenated by the linker as if they had a PUBLIC combine type. The only difference is that for executable task image formats that support initialization of SS:ESP, the linker sets the initial values of the SS and ESP registers to point to the end of the segment.
COMMON	Segments with the same name having COMMON combine type are overlaid. Each of the component segments is relocated to the same address: the base of the composite segment. The length of the composite segment is the length of the biggest component. COMMON segments can have initialized data, although care should be taken not to initialize the same area in a COMMON segment to different values in two different modules.
MEMORY	The MEMORY combine type is a synonym for the PUBLIC combine type. This type is provided for Intel compatibility.
AT *address*	The AT specifier tells the assembler and the linker that the segment has a paragraph base of *address* when the program is executed. The *address* parameter can be any expression which evaluates to an absolute number. AT segments are intended for use with programs that are burned into PROM. These programs are generally linked in some absolute format such as Intel hex format. For programs which are linked as a relocatable task image (e.g., MS-DOS .EXE format), an AT segment cannot contain initialized data and is useful only as a template for data already in memory. An example of such an area is the screen memory buffer on an IBM-PC.

If the AT combine type is given for a segment, it must be the last segment attribute specifier on the line. Note also that an AT combine type implies a PARA align type. For this reason, no align type should be specified for a segment defined with an AT combine type.

If a segment's combine type is not specified when it is defined, it is given a default combine type of private and is not combined with any other segments.

3.7.3 Use Attribute Specifier

The use attribute specifier of a segment informs the assembler what value it should expect the D bit in the segment's descriptor table entry (please see the list of related documents in the Preface) to be during execution. The D bit is used by the 80386 CPU in protected mode to decide whether the default operand and address mode sizes are 16 or 32 bits wide.

The assembler needs to know what value the D bit will have during program execution in order to correctly generate address size and operand size override bytes. The use attribute specifier can have one of the following two values:

Use Attribute	Meaning
USE16	The segment being created has a default operand and address size of 16 bits. All offsets for instruction and data references default to 16 bits, and instruction operands are given a default size of 16 bits. The segment being created is limited to a maximum size of 64K bytes.
USE32	The segment being created has a default operand and address size of 32 bits. All offsets for instruction and data references default to 32 bits, and instruction operands are given a default size of 32 bits.

If the segment's use attribute is not specified, a default is assigned based on the use attribute of its enclosing segment and the target CPU of the assembly. If the segment is nested inside another segment, it will be given the use type of the enclosing segment. If it is not nested, it will be assigned a default of USE32 if the target CPU is an 80386. Otherwise, a default of USE16 is assigned.

Programs executing on the 8086/8088/80186/80188/80286 processors, or in real mode on the 80386, can only have USE16 segments. Programs executing in protected mode on the 80386 can have either USE16 or USE32 segments.

Please see also: Chapter 11, Appendix J

3.7.4 Access Type Specifier

The access type specifier defines how the segment can be accessed during program execution. It has any of the following values:

Access Type	Meaning
RO	The segment is a read only data segment.
BO	The segment is an execute only code segment.
ER	The segment is a readable code segment.
R W	The segment is a writable data segment.

If no access type is specified for the segment, a default of ER is assigned. An access type is only specified if the target CPU is an 80286 or an 80386. 8086, 8088, 80186, and 80188 CPUs do not support segment protection.

3.7.5 Class Specifier

The class specifier for a segment defines the name of the class of which it is a member. The class name is enclosed in single quotation marks and may be any length. If no class specifier is given for a segment, a default of the null class name is assigned.

386 | LINK uses a segment's class name as part of its algorithm for deciding how to order segments in memory. Segments with the same class name are ordered contiguously in memory. If a segment is a member of the null class, it is ordered contiguously with all other segments which are also members of the null class.

3.8 ENDS Directive

Syntax:

```
segname ENDS
```

The ENDS directive terminates the current definition of *segname* and causes assembly to continue into the enclosing segment, if any. The current location counter value for the segment is saved and assembly continues again at that point if the segment is reopened.

The *segname* parameter must be the name of the most recently opened segment. If it is the name of a different segment, a block nesting error occurs and no segment is closed.

Note that the ENDS directive is also used to terminate a structure definition (please see section 6.6.2). The assembler uses the *segname* parameter to determine which usage of ENDS is intended.

Example:

The following example shows code, data, and stack segments created for a module being assembled for an 8086 target CPU:

```
dseg    SEGMENT word public 'data'
          .
          .
dseg    ENDS

cseg    SEGMENT word public 'code'
          .
          .
cseg    ENDS

sseg    SEGMENT word stack 'stack'
          .
```

```
        .
sseg    ENDS
```

This example demonstrates correctly nested segments:

```
cseg    SEGMENT
        .
        .
dseg    SEGMENT
        .
        .
dseg    ENDS
        .
        .
cseg    ENDS
```

A segment which can be overlaid on the screen memory of an IBM-PC with a CGA display controller board looks like this:

```
scrseg  SEGMENT at 0B800h
        .
        . ; Template for screen memory goes here.
        .
scrseg  ENDS
```

Finally, an example of some 16- and 32-bit 80386 segments:

```
dseg    SEGMENT dword public use32 'data'
        .
        .
dseg    ENDS

cseg16   SEGMENT word public use16 'code16'
        .
        .
cseg16   ENDS

cseg32   SEGMENT dword public use32 'code32'
        .
        .
cseg32   ENDS
```

Please see also: SEGMENT (3.7) , ENDS (6.6.2)

3.9 GROUP Directive

Syntax:

```
groupname GROUP segname,...
```

The GROUP directive is used to define a group of one or more segments, causing all segments in the group to be relocated relative to the base address of the first segment in the group. The *groupname* parameter defines the name of the group and must be unique. The *segname* parameter must be the name of a segment defined in the module.

Note that a group definition does not affect how the segments are ordered in memory. Ordering is done with the class specifier of each segment. The group directive is only used to establish the fact that several segments are accessed with one segment register and should thus be relocated relative to the same segment base. It is possible to have segments which are not in a group located between those which are in a group. The only restriction is that the end of the last segment in the group must not be more than four gigabytes (for an 80386 link) or 64K bytes (for a link with any other target CPU) from the beginning of the first segment in the group. This problem should never occur in an 80386 link, but it is signaled by fixup overflow errors in an 8086 link.

Example:

Most C programs have several segments for data which are combined into one group. One is for constant data, one is for variables, and one is for the stack. At runtime, however, all three segments are accessed from the DS segment register. The following code is used to inform 386|ASM of this program structure, so it can generate correct relocation information for the linker:

```
const   SEGMENT word public 'CONST'
          .
          .
const   ENDS

data    SEGMENT word public 'DATA'
          .
          .
data    ENDS
```

```
stack   SEGMENT word stack 'STACK'
        .
        .
stack   ENDS

dgroup  GROUP const, data, stack
        ASSUME ds:dgroup
```

Please see also: SEGMENT (3.7), ASSUME (3.10), Chapter 11

3.10 ASSUME Directive

Syntax:

```
ASSUME regname:segname|regname:NOTHING,...
ASSUME NOTHING
```

The ASSUME directive is used to inform the assembler what values the segment registers have during program execution. The *regname* parameter must be the name of one of the segment registers: CS, DS, ES, FS, GS, SS. (FS and GS are allowed only when the assembly is targeted for an 80386 CPU.) *Segname* must be the name of a segment or group defined in the module being assembled. The keyword NOTHING can be used instead of the *segname* parameter to indicate that the register will not point at any segment in the program. It can also be used instead of the *regname* parameter as shorthand to say that all segment registers contain undefined values. If the same register name is used more than once in an ASSUME directive, no error is signaled and the last (rightmost) value assumed is the one used.

The assembler uses the segment register assumptions to establish addressability of data and instruction references and to generate segment override bytes where necessary. It is to the programmer's advantage to be as complete as possible in telling the assembler what values the segment registers have whenever they change.

Example:

```
; Assume CS points at the program's code and DS points
; at nothing.
ASSUME cs:code,ds:nothing
```

```
; No assumptions are valid.
ASSUME nothing

; Assume CS points at the program's code and all other
; registers point at the data group.
ASSUME cs:code,ds:dgroup,es:dgroup,ss:dgroup
ASSUME fs:dgroup,gs:dgroup
```

Please see also: SEGMENT (3.7), GROUP (3.9), Chapter 11

3.11 Location Counter Control Directives

The assembler maintains a location counter for each segment defined in a module. The location counter is the offset where the next instruction or data bytes are assembled into the segment. 386|ASM provides several directives for changing the current value of the location counter if necessary.

3.11.1 ORG Directive

Syntax:

```
ORG expression
```

The ORG directive is used to set the current value of the location counter. The *expression* parameter is any expression which evaluates to an absolute number or an offset in the current segment. The ORG directive only affects the location counter in the currently open segment. The location counters for any other segments (including enclosing segments) are not affected.

After the ORG directive is processed, any instruction or data bytes emitted in the current segment are placed at the location specified by *expression*.

Example:

```
; Set the loc. ctr. to be offset 100h into the segment.
      ORG 100h

; Create an array which can be accessed two different
; ways depending on which name is used.
```

```
        barray DB 100 dup(?)
        ORG barray
        warray DW 50 dup(?)

; Origin the location counter 10 bytes into barray.
        ORG barray+10
```

Please see also: EVEN (3.11.2), ALIGN (3.11.3)

3.11.2 EVEN Directive

Syntax:

```
EVEN
```

The EVEN directive is used to force alignment on a word boundary of the next instruction or data byte generated. It does so by emitting a NOP instruction if the current value of the location counter is odd. If the location counter is even, this directive has no effect.

The EVEN directive cannot be used on segments which have a BYTE align type because the assembler cannot know whether the segment is going to have an even or odd base address when 386 | LINK combines it with other segments.

Example:

The following example demonstrates how to word align an array of 100 words:

```
EVEN
warray  DW 100 dup(?)
```

This next example forces a NOP byte to be generated before the MOV instruction:

```
cseg     SEGMENT use16
         ORG 100h
         sub ax,1000h
         EVEN
         mov bx,ax
cseg     ENDS
```

Please see also: ORG (3.11.1), ALIGN (3.11.3)

3.11.3 ALIGN Directive

Syntax:

```
ALIGN expression
```

The ALIGN directive is used to align the location counter of the currently open segment to the alignment boundary specified by *expression*. The *expression* is required to be a power of two, and an error is signaled if it is not.

386 | ASM does not verify that the specified alignment value makes sense for the segment's alignment type. It does not make sense to align the location counter for a segment to a value larger than the granularity of the segment's align type, because the assembler does not know how the segment will be combined with other segments by 386 | LINK.

If the number of bytes needed to align the location counter on the specified boundary is less than four, 386 | ASM uses NOP bytes to fill the space. If the number of bytes is four or more, 386 | ASM simply adjusts the location counter, causing the space to be filled with zero bytes by 386 | LINK. This feature is useful for aligning instructions in a code segment on word or double word boundaries.

Example:

```
;
; Align the location counter on a double word boundary
;
   ALIGN 4

;
; Align the location counter on a paragraph boundary.
;
   ALIGN 16

;
; Align the location counter on a page boundary.
;
   ALIGN 256
```

Please see also: ORG (3.11.1), EVEN (3.11.2)

Chapter 4

Defining Constants with the EQU and = Directives

4.1 Introduction

It is frequently desirable to substitute symbolic names for the constant values used during an assembly. They make code more readable and easier to support. 386 | ASM supports three types of symbolic constants: integer values, aliases for assembler keywords, and strings of arbitrary text.

Symbolic constants are defined using the EQU and equal sign (=) directives. The EQU and equal sign directives can also be used to define variables and labels (please see section 5.2.3 and section 6.5.3 for details).

4.2 EQU Directive

Syntax:

```
name EQU expression
```

The EQU directive is used to define all three types of constants: integer values, aliases, and text strings. The *name* parameter specifies the symbol's name, which must be unique. The assembler replaces each subsequent occurrence of *name* with its constant value, whether it is an integer, an alias, or a string of text.

Symbols defined with the EQU directive cannot be redefined.

4.2.1 Using EQU to Create Absolute Constants

Syntax:

```
name EQU expression
```

The assembler first attempts to evaluate the *expression* following the EQU directive as an integer expression. If it evaluates to an absolute constant, *name* is entered in the user-defined symbol table as an absolute constant symbol. An absolute constant is not relocatable, does not have a data type, and does not reference any undefined or forward defined symbols. If the expression is an address expression (relocatable), *name* is entered in the user-defined symbol table as a label or variable. Consult sections 5.2.3 and 6.5.3 for details.

An absolute constant symbol can be used anywhere an integer is valid in an expression. Its value can also be made available to other modules in a program using the PUBLIC directive.

4.2.2 Creating Text Substitution Symbols

Syntax:

```
name EQU arbitrary text
```

If the assembler is unable to successfully evaluate the remainder of the line which follows the EQU directive as an integer constant, it saves it as a text string instead. Each subsequent occurrence of *name* is replaced with the *arbitrary text* which follows its definition. This is a simple way of creating macros which have no parameters.

Note that a text constant is evaluated by the assembler each time it is expanded. Thus, its value can change each time it is used if the value of one of the symbols in the string has changed. Care must be taken when defining and using text strings if a constant value is desired.

4.2.3 Creating Aliases for Assembler Keywords

Syntax:

```
name EQU keyword
```

An alias for an assembler keyword is actually a special case of a text substitution. If the arbitrary text which follows the EQU directive is a single assembler keyword, *name* is entered in the user-defined symbol table as a reference to the *keyword* instead of a text string. This form of

substitution requires less symbol table space and is expanded more quickly by the assembler.

4.2.4 Examples with EQU

Example:

```
;
; define some constants with EQU
;
c1      EQU 10h         ; absolute constant = 16
c2      EQU c1+10       ; absolute constant = 26
t1      EQU word ptr    ; t1 is a text string
arg1    EQU 8[ebp]      ; arg1 is a text string
t2      EQU c3+5        ; t2 is text because of
                        ; forward reference
c3      EQU 10h*10h     ; absolute constant = 256
stk_frm EQU ebp         ; alias for keyword EBP

;
; use constants defined with EQU in instructions
;
mov     eax,c1         ; load 16 into EAX
mov     ebx,c2         ; load 26 into EBX
mov     ecx,arg1       ; load 8[ebp] into ECX
mov     edx,t2         ; load 261 into EDX
mov     esi,c3         ; load 256 into ESI
push    stk_frm        ; set up
mov     stk_frm, esp   ; stack frame
```

Please see also: = (4.3), EQU (5.2.3), EQU (6.5.3), PUBLIC (7.2),-DEFINE (2.2.7)

4.3 The = (Equal Sign) Directive

Syntax:

```
name = expression
```

The equal sign directive is used to create constants in much the same way as the EQU directive, with the restriction that it cannot create text substitution or alias symbols.

Absolute constants defined with the equal sign directive can be used in an expression anywhere an integer is valid. Their values can also be made available to other modules using the PUBLIC directive.

The difference between constants defined with the EQU directive and constants defined with the equal sign directive is that constants defined with the equal sign directive can be redefined at any time by using their name with another equal sign directive. The assembler replaces each subsequent occurrence of *name* with its most recent value when the reference is encountered.

Example:

```
c1 = 10h            ; absolute constant = 16
mov      ax,c1      ; load 16 into AX
c1 = 20h            ; absolute constant = 32
mov      bx,c1      ; load 32 into BX
```

Please see also: EQU (4.2), = (5.2.3), = (6.5.3), PUBLIC (7.2)

Instruction Labels, Control Transfer, and Procedure Blocks

5.1 Introduction

Most control transfer instructions in the 80386 instruction set require an instruction label as their operand. An instruction label is a user-defined symbol which has an address and a data type. Its address has two attributes: a segment and an offset. The segment attribute is the segment in which the label resides, and the offset attribute is the location of the label within its segment. A label's data type must be either NEAR or FAR and specifies how the assembler expects control to be transferred to it. The assembler uses this information to generate the correct instruction opcode for CALLs or JMPs to a label, and for a RET within a procedure. A NEAR label is reached with an intra-segment transfer, while a FAR label is reached with an inter-segment transfer. Some control transfer instructions only support intra-segment transfers; consult Appendix D for the valid address modes for a specific instruction. If you are not familiar with the NEAR and FAR control transfers dictated by the segmented architecture of the 8086 processor family, read Chapter 11 before reading this chapter.

Instruction labels and procedure names may be made globally accessible with the PUBLIC and EXTRN directives.

5.2 Instruction Labels

There are several ways to create instruction labels in the user-defined symbol table. Each method is described in detail below.

5.2.1 Creating a Label with : (Colon)

Syntax:

```
labelname:
```

The simplest way to create an instruction label is to follow the label name with a colon (:). This creates a label named *labelname* which has a data type of NEAR, a segment attribute of the currently open segment, and an offset attribute equal to the current value of the location counter. The label name must be the first thing that appears on the source line, and it must be unique.

There is no way to create a label with a data type of FAR using : .

Example:

The example below shows how control can be transferred to one of two NEAR labels depending on whether the accumulator contains the value 4:

```
        cmp     eax,4
        je      equal4
        jmp     nequal4

equal4:

nequal4:
```

Please see also: PUBLIC (7.2)

5.2.2 Creating a Label Using the LABEL Directive

Syntax:

```
labelname LABEL type
```

The LABEL directive creates a label named *labelname* in the user-defined symbol table. The *type* parameter specifies the label's data type and may be either NEAR or FAR. Like labels created with :, a label created with the LABEL directive has a segment attribute of the currently open segment and an offset attribute equal to the current value of the location counter.

Example:

The following example demonstrates how to define two entry points to the same routine, with one having a data type of NEAR, and the other having a data type of FAR:

```
nearentry   LABEL near     ; Near entry point
farentry    LABEL far      ; Far entry point

            mov eax,5     ; Subroutine begins here
                    .
                    .
                    .
```

Please see also: LABEL (6.5.2), PUBLIC (7.2)

The LABEL directive can also be used to create variables. A description of the syntax appears in section 6.5.2.

5.2.3 Creating a Label Using the EQU and Equal Sign Directives

Syntax:

```
labelname EQU address expression
labelname = address expression
```

If the expression which follows an EQU or an equal sign directive evaluates to a relocatable quantity having a data type of NEAR or FAR, 386 | ASM enters the symbol in the symbol table as a label instead of a constant. Using this method, it is possible to create a label with arbitrary segment and offset attributes. The *address expression* can be any expression which evaluates to a relocatable quantity having a data type of NEAR or FAR. The new label has a segment attribute equal to the segment attribute of the expression and an offset attribute equal to the expression's value.

Example:

Each of the following examples use the EQU directive to create instruction labels:

```
cseg1   SEGMENT   word public 'code'
l1      LABEL     far
cseg1   ENDS
```

```
cseg2   SEGMENT   word public 'code'

lab1    EQU       $     ; $ has a data type of AR
lab2    EQU       $+5   ; 5 bytes past the loc. r.
lab3:

; A far label at the same address as lab3.
lab4    EQU far ptr lab3
; A far label 7 bytes past lab4
lab5    EQU lab4+7
; A near label at the current loc. ctr.
lab6    EQU THIS near
; A far label in segment cseg1 4 bytes past l1
lab7    EQU l1+4
cseg2   ENDS
```

Please see also: EQU (4.2), EQU (6.5.3), = (4.3), = (6.5.3), PUBLIC (7.2), $ (9.2.2), THIS (9.3.13)

The EQU and equal sign directives can also be used to create variables. If the address expression following the directive has the data type of a variable, a variable is created. Please see section 6.5.3 for details.

5.2.4 Control Transfers to Address Expressions

Control transfer instructions normally require an instruction label as their operand. However, it is permissible to use an address expression with data type NEAR or FAR. (Some instructions only permit a data type of NEAR.) Thus, it is possible to transfer control to the location specified by any address expression without creating a label located at the destination.

While this practice is discouraged for reasons of style, there may be cases where such expressions are useful.

Example:

```
cseg1    SEGMENT   word public 'code'
lab1     EQU       $
; a very simple address expression
         jmp       lab1
; 5 bytes past lab1
         jmp       lab1+5
```

```
; a far jump to a near label
          jmp       far ptr nearlab
cseg1     ENDS
cseg2     SEGMENT   word public 'code'
nearlab   LABEL     near
cseg2     ENDS
```

5.2.5 Control Transfers Using Indirection

The CALL and JMP instructions also allow a variable as their operand. When the assembler sees a CALL or JMP instruction with a variable as its operand, it generates an indirect form of the instruction. When the instruction is executed, control is transferred to the location pointed to by the variable. The type of transfer generated (NEAR or FAR) depends on the data type of the variable and the use type of the currently open segment. In most cases, a variable of the same size as the use attribute of the currently open segment generates a NEAR transfer, and a variable two bytes larger generates a FAR transfer. There are exceptions to this rule due to certain ambiguities; consult Appendix J for a full explanation.

Example:

Assuming we are assembling instructions for an 8086 target, the following instructions generate indirect control transfers:

```
label1  LABEL near
nptr    DW      label1
fptr    DD      label1

call    nptr          ; Near transfer (16-bit ptr)
call    bx            ; Near transfer (16-bit reg)
call    fptr          ; Far transfer (32-bit ptr)
```

And if we are in a USE32 segment, assembling instructions for an 80386 target:

```
label2  LABEL near

nptr    DD      label2
fptr    DF      label2

call    nptr          ; Near transfer (32-bit ptr)
call    ebx           ; Near transfer (32-bit reg)
call    fptr          ; Far transfer (48-bit ptr)
```

5.3 Procedure Blocks

Procedure blocks are used to split a program into logical sections. Typically, each subroutine in a program is given its own procedure block. The name of the subroutine is given to the procedure block. The name of a procedure block has the same attributes as a label and can be used any place an instruction label is valid in an expression.

A procedure block definition is started with the PROC directive and is terminated with the ENDP directive. Normally, a procedure will contain at least one RET instruction to return control to the caller. The assembler generates the form (NEAR or FAR) of the RET instruction that corresponds to the data type of the procedure.

A procedure block can also contain an arbitrary number of locally defined symbols. A locally defined symbol is only visible inside the procedure block in which it is defined, thus permitting local symbols to have the same names in different procedure blocks.

5.3.1 PROC Directive

Syntax:

```
procname PROC {type}
```

The PROC directive marks the beginning of a procedure block. The *procname* parameter specifies the name of the procedure, which must be unique. The optional *type* parameter specifies the type of the procedure, which must be either NEAR or FAR. If no procedure type is specified, a default type of NEAR is used. A procedure's address can be made available to other modules in a program with the PUBLIC directive.

Nesting of procedure definitions is permitted, but the nesting must be strictly block structured. In other words, the definitions cannot overlap.

Example:

The example below demonstrates a procedure which loads the number 5 into the AX register and returns to its caller:

```
load5 PROC near

      mov ax,5
      ret

load5 ENDP
```

This example demonstrates a procedure correctly nested inside another procedure:

```
outer PROC near     ; Start of outer procedure
.
.
inner PROC near     ; Start of nested procedure
.
inner ENDP          ; End of nested procedure
.
.
outer ENDP          ; End of outer procedure
```

Finally, an example of incorrectly nested procedures:

```
outer PROC near      ; Start of outer procedure

inner PROC near      ; Start of nested procedure

outer ENDP           ; ERROR - outer ends before

inner ENDP           ; ERROR
```

Please see also: ENDP (5.3.2), PUBLIC (7.2)

5.3.2 ENDP Directive

Syntax:

```
procname ENDP
```

The ENDP directive terminates the definition of a procedure block. The *procname* parameter specifies the name of the procedure to be closed, which must be the name of the most recently opened procedure. If *procname* is not the name of the most recently opened procedure, an error is signaled and the most recently opened procedure is then closed.

It is permissible to close a segment without closing a currently open procedure. However, the segment must be reopened and the procedure block closed before the end of the source file. It is illegal to close a procedure in a different segment from the one in which it was opened.

Please see also: PROC (5.3.1)

5.3.3 Local Symbol Definitions Within Procedure Blocks

When inside a procedure block, all variable, label, and constant definitions which use the pound character (#) as the first character of the symbol's name are processed as local symbol definitions. A local symbol definition is only visible inside the procedure in which it is defined, and the symbol name need only be unique to that procedure. It is not visible to any enclosing procedures, nor is it visible to any procedures nested inside the procedure being defined. The syntax of a local symbol definition is the same as that of a normal symbol definition, except that the name of the symbol being defined must start with the # character.

Local symbols are a convenient way to create temporary labels and variables without having to give each one a unique name. Such symbols are useful as labels for conditional jumps, temporary variables, and offsets of function parameters in the runtime stack.

Note that a local symbol takes up the same amount of space in the symbol table as a normal symbol. Its definition remains in the symbol table even after the enclosing procedure is terminated; it just becomes invisible.

Local symbols cannot be external, nor can their values be made available to other modules with the PUBLIC directive.

Example:

The following example demonstrates the use of local symbols within a procedure block. It creates and uses a local constant, a label, and a variable.

```
proc1  PROC near

#arg1 = 6

       mov      eax,#arg1[esp]
```

```
        or      eax,eax
        jnz     #l1
        ret
#l1:
        mov     CS:#temp1,eax
        ret

#temp1 DD 0

proc1   ENDP
```

Please see also: Labels (5.2), Variables (6.5), Defining Constants With the EQU and = Directives (Chapter 4), Symbol Formation (3.2.2)

Chapter 6

Variables and Data Declarations

6.1 Introduction

The data declaration directives are used to create and optionally initialize blocks of memory in a segment. In addition to providing directives which initialize any of the assembler–defined data types, 386 | ASM also allows the programmer to define and initialize arbitrary data structures. The assembler–defined data types are initialized using the DB, DW, DD, DF, DP, DQ, and DT directives, and user–defined data types can be created using the STRUC and RECORD directives. Variables may be made globally accessible with the PUBLIC and EXTRN directives.

The data declaration directives are also used to create variables. Like an instruction label, a variable is a user–defined symbol which has an address and a data type. Its address has two components: a segment attribute and an offset. The segment attribute is the segment in which the variable resides, and the offset attribute is the location of the variable within its segment. A variable's data type is either one of the assembler–defined data types or the name of a user-defined structure or record.

The assembler usually must know the data type of instruction operands in order to choose the correct form of the instruction. If a variable is used in an expression for an operand, the operand is given the data type of the variable. This default can be overridden if necessary by using the PTR operator.

6.2 Data Declaration Directives

The following sections describe the data declaration directives for the assembler-defined data types in more detail. The same syntax is used for all the data declaration directives, however, each directive allows a slightly different set of values in its initializer list.

6.2.1 DB Directive

Syntax:

```
{name} DB value,...
```

The DB directive is used to initialize one or more bytes in the currently open segment from the list of values which follows the directive on the line. If the optional *name* is specified, a variable of type BYTE is created in the user-defined symbol table. It will have a segment attribute of the currently open segment and an offset equal to the current value of the location counter.

The *value* parameter is an expression which evaluates to an integer, a string constant, or the undefined storage operand (?). If the value is an integer, one byte is allocated and initialized to the value of the expression. If the value is the undefined storage operand, a byte is allocated but not given an initial value. If the value is a string constant, one byte is allocated and initialized for each character in the string. Relocatable values are permitted with the DB directive, though it is likely that a fixup overflow will occur when 386 | LINK attempts to resolve the relocatable reference and to fit it into a byte.

If there is more than one value on the line following the directive, each value must be separated by a comma. 386 | ASM then allocates and initializes as many bytes as necessary to process it.

Example:

```
factor  DB 10h
astring DB 'This is an ASCII string'
        DB 'A NULL terminated string',0
intexpr DB 12+4*4
listval DB 1,2,3,4,5
unknown DB ?
```

Please see also: DUP (6.3), Undefined Storage Operand (?) (6.4),PUBLIC (7.2), String Constants (3.2.3.4)

6.2.2 DW directive

Syntax:

```
{name} DW value,...
```

The DW directive is used to initialize one or more words (a word is two bytes) in the currently open segment from the list of values which follows the directive on the line. If the optional *name* is specified, a variable of type WORD is created in the user-defined symbol table. It will have a segment attribute of the currently open segment and an offset equal to the current value of the location counter.

The *value* parameter is an expression which evaluates to an integer, a one- or two-character string constant, or the undefined storage operand (?). If the value is an integer or a character string constant, one word is allocated and initialized to the value of the expression. If the value is the undefined storage operand, a word is allocated but not given an initial value. Relocatable values are permitted with the DW directive, though it is possible that a fixup overflow will occur when 386 | LINK attempts to resolve the relocatable reference.

If there is more than one value on the line following the directive, each value must be separated by a comma and each will allocate and initialize its own word of memory.

Example:

```
factor  DW 1000h
char1   DW 'A'
char2   DW 'AB'
        DW 12+4*4
listval DW 1,2,3,4,5
relval  DW OFFSET factor
unknown DW ?
```

Please see also: DUP (6.3), Undefined Storage Operand (?) (6.4), PUBLIC (7.2)

6.2.3 DD Directive

Syntax:

```
{name} DD value,...
```

The DD directive is used to initialize one or more double words (a double word is four bytes) in the currently open segment from the list of values which follows the directive on the line. If the optional *name* is specified, a variable of type DWORD is created in the user-defined symbol table. It has a segment attribute of the currently open segment and an offset equal to the current value of the location counter.

The *value* parameter is an expression which evaluates to an integer, a character string constant from one to four characters in length, a four-byte real number, a four-byte encoded real number, or the undefined storage operand (?). If the value is an integer or a character string constant, a double word is allocated and initialized to the value of the expression. If the value is the undefined storage operand, a double word is allocated but not given an initial value. If the value is a real number, four bytes are allocated and the value is stored as an 80287 format short real.

If *value* is relocatable, 386 | ASM outputs different object code depending on the target CPU of the assembly. If the target CPU is an 80386, the relocatable expression is output as a 32 bit offset which will be resolved by the linker. If the target CPU is an 8086, 80186, or 80286, a relocatable value is output as a 16 bit offset followed by a 16 bit segment selector. These values will be resolved by the linker to the offset portion of the value expression followed by the segment portion of the value expression.

If there is more than one value on the line following the directive, each value must be separated by a comma, and each will allocate and initialize its own double word of memory.

Example:

```
factor   DD 10000h
char1    DD 'A'
char2    DD 'ABCD'
         DD 10000h*10h+07Fh
encreal  DD 3F800000r ;1.0
realval  DD 1.0E-10
```

```
listval DD 1,2,3.14159,4,5
reloc1  DD factor
reloc2  DD OFFSET factor
unknown DD ?
```

Please see also: DUP (6.3), Undefined Storage Operand (?) (6.4), PUBLIC (7.2)

6.2.4 DF and DP Directives

Syntax:

```
{name} DF value,...
{name} DP value,...
```

The DF and DP directives (which are synonyms and can be used interchangeably) are used to initialize one or more 48 bit pointer words in the currently open segment from the list of values which follows the directive on the line. If the optional *name* is specified, a variable of type PWORD is created in the user-defined symbol table. It will have a segment attribute of the currently open segment and an offset equal to the current value of the location counter.

The *value* parameter can be a six-byte integer, a character string constant from one to six characters in length, a relocatable quantity, or the undefined storage operand. If the value is a six-byte integer or string constant, six bytes are allocated and initialized to the value specified. If it is the undefined storage operand, six bytes are allocated but not initialized.

If the value is a relocatable quantity, it is output as a 32 bit offset followed by a 16 bit segment selector. These values will be resolved by 386|LINK to the offset portion of the relocatable expression followed by the segment portion of the expression.

If there is more than one value on the line following the directive, each value must be separated by a comma, and each will allocate and initialize its own six byte piece of memory.

The DF and DP directives are only available if the 80386 is the target CPU of the assembly.

Example:

```
sixbyte DP 123456789ABCh
char1   DP 'A'
        DP 'ABCDEF'
reloc1  DF sixbyte
listval DF 1,2,char1,4,5,sixexpr
unknown DF ?
```

Please see also: DUP (6.3), Undefined Storage Operand (?) (6.4), PUBLIC (7.2)

6.2.5 DQ Directive

Syntax:

```
{name} DQ value,...
```

The DQ directive is used to initialize one or more quad words (a quad word is eight bytes) in the currently open segment from the list of values which follows the directive on the line. If the optional *name* is specified, a variable of type QWORD is created in the user–defined symbol table. It will have a segment attribute of the currently open segment and an offset equal to the current value of the location counter.

The *value* parameter can be an eight byte integer, a character string constant from one to eight characters in length, an eight byte real number, an eight byte encoded real number, or the undefined storage operand (?). If the value is an integer or a character string constant, a quad word is allocated and initialized to the value specified. If the value is the undefined storage operand, a quad word is allocated but not given an initial value. If the value parameter is a real number, eight bytes are allocated, and the value is stored as an 80287 format long real.

If there is more than one value on the line following the directive, each value must be separated by a comma, and each will allocate and initialize its own quad word of memory.

Example:

```
quadint DQ 123456789ABCDEF0h
char1   DQ 'A'
char2   DQ 'ABCDEFGH'
```

```
realval DQ 1.0E200
encreal DQ 3FF0000000000000r ;1.0
        DQ 1,2,3.14159,4,5
unknown DQ ?
```

Please see also: DUP (6.3), Undefined Storage Operand (?) (6.4), PUBLIC (7.2)

6.2.6 DT Directive

Syntax:

```
{name} DT value,...
```

The DT directive is used to initialize one or more ten byte chunks of memory in the currently open segment from the list of values which follows the directive on the line. If the optional *name* is specified, a variable of type TBYTE is created in the user–defined symbol table. It will have a segment attribute of the currently open segment and an offset equal to the current value of the location counter.

The *value* parameter can be a ten byte integer, a character string constant from one to four characters in length, a ten byte real number, a ten byte encoded real number, a ten byte packed decimal number, or the undefined storage operand (?). If the value is an integer or a character string constant, ten bytes are allocated and initialized to the value specified. If the value is the undefined storage operator, ten bytes are allocated but not initialized. If the value is a real number, ten bytes are allocated, and the value is stored as an 80287 format temporary real. Lastly, if the current default radix is base 10 and there is no radix specifier at the end of the value, it is stored as a ten byte 80287 packed decimal number.

If there is more than one value on the line following the directive, each value must be separated by a comma, and each will allocate and initialize its own ten byte piece of memory.

Note that the DT directive assumes that a constant which is composed entirely of decimal digits is a packed decimal number when the default radix is base 10. If you wish to use the DT directive to define a ten-byte integer value, you must follow the digits with the "D" radix specifier to indicate that it is an integer composed of decimal digits.

Example:

```
tbyteint DT 123456789ABCDEF01234h
char1    DT 'A'
         DT 'ABCD'
realval  DT 1.0E1000
encreal  DT 3FFF8000000000000000r ;1.0
packdec  DT 12345678901234567890
listval  DT 1,2,3.14159,4,5
unknown  DT ?
```

Please see also: DUP (6.3), Undefined Storage Operand (?) (6.4), PUBLIC (7.2)

6.3 DUP Operator

Syntax:

```
count DUP (value,...)
```

The DUP operator is a special purpose operator which is used to create multiple instances of a given initial value or list of values. The *count* parameter specifies how many instances of the value list should be created. It can be any expression which evaluates to an absolute integer value. Forward references to symbols are illegal in the expression.

The *value* parameter (or list of parameters) must be enclosed by parentheses and can be any valid expression or another DUP operator. If more than one value is specified, each element in the list must be separated by a comma. The maximum permitted nesting level of DUP operators is 17.

Example:

The following examples demonstrate the use of the DUP operator with the data declaration directives to initialize multiple copies of values:

```
        DB 500 DUP(?)
length  equ 1000h
warray  DW length DUP(0)
stack   DB 1024 DUP('STCK')
alist   DD 100 DUP(1,2,3,4)
```

```
nest1      DT 10 DUP(5 DUP(1.0))
nest2      DB 5 DUP(1,2 DUP(?))
```

Please see also: Undefined Storage Operand (?) (6.4)

6.4 Undefined Storage Operand (?)

The undefined storage operand, denoted by "(?)", is used to indicate to the assembler that the value a location has at assembly time does not matter, so no initial value is specified for it. It can be used as a value for any of the data declaration directives, but cannot be combined with other values in an expression.

The undefined storage operand is most often used with the DUP operator to allocate large pieces of memory without initializing them to a value.

Example:

```
DB ?              ; an undefined byte value
DD 1,2,?,4,5      ; an undefined double word in an array
DW 500 DUP(?)     ; an array of 500 uninitialized words.
```

Please see also: DUP (6.3)

6.5 Creating Variables

A variable is a user-defined symbol which has an address attribute and a data type. A variable is not required to occupy any space; thus two variables can reference the same block of memory. This section describes the different ways to create variables which have an assembler-defined data type. The method for creating variables with a user–defined data type is described in sections 6.6-6.11, which cover structure and record declarations.

6.5.1 Creating a Variable with a Data Declaration Directive

Each of the data declaration directives can create a variable by preceding the directive with the name of the variable. This is a simple way to create a variable which requires that space be allocated for it. Variables created

with the data declaration directives have a segment attribute of the currently open segment and an offset attribute equal to the value of the location counter at the time the directive is processed.

Example:

The following examples demonstrate how to create variables using the data declaration directives. The examples use simple values for the value list which follows the directive. A complete description of allowable values appears in the section describing each directive.

```
bytevar  DB ?              ; An undefined byte value.
wordvar  DW 14+15          ; A word with value 29.
dwordvar DD 1.4            ; A dword variable with
                           ; a short real value.
tbytelst DT 1,2,3,4,5      ; A list of 5 tbyte
                           ; values.
bytearr  DB 1024 DUP(?)    ; A 1024 byte array.
```

Please see also: DUP (6.3), Undefined Storage Operand (?) (6.4), PUBLIC (7.2)

6.5.2 Creating a Variable with the LABEL Directive

Syntax:

```
name LABEL type
```

The LABEL directive can also be used to create variables using the syntax shown above. The *name* parameter is the name of the variable to be created and the *type* parameter is any of the assembler-defined data types. Variables created with the LABEL directive have a segment attribute of the currently open segment and an offset attribute equal to the value of the location counter when the directive is processed.

The location counter is not changed when the LABEL directive is processed, so variables defined with it occupy no space. The LABEL directive is a convenient way to create two variables which can be used to address the same block of memory in different ways.

Example:

```
var1    LABEL word      ;  A word variable.
var2    LABEL dword     ;  A double word variable.

; An array which can be accessed as a byte or word
; array depending on which name is used.

barray  LABEL byte
warray  DW 512 DUP(?)
```

Please see also: LABEL (5.2.2), PUBLIC (7.2)

6.5.3 Creating a Variable with the EQU and Equal Sign (=) Directives

Syntax:

```
name EQU address expression
name  =  address expression
```

If the expression which follows an EQU or equal sign (=) directive evaluates to a relocatable quantity having one of the assembler-defined data types, 386|ASM enters the symbol in the symbol table as a variable instead of a constant. Using this method, it is possible to create a variable with arbitrary segment and offset attributes. The *name* parameter is the name the variable has, which must be unique. The *address expression* can be any expression which evaluates to a relocatable quantity and has an assembler–defined data type. The variable has a segment attribute equal to the segment attribute of the expression.

Example:

The following examples use the EQU and = directives to create variables:

```
var1    EQU    word ptr $     ; A word at the loc. ctr.
var2    EQU    byte ptr var1 ; A byte at the same
                             ; address as var1.
foo     DB     5
var3    EQU    foo+5         ; A byte 5 past foo.
var4    =      THIS dword    ; A dword at the loc. ctr.
```

Please see also: EQU (4.2), EQU (5.2.3), = (4.3), = (5.2.3), PUBLIC (7.2), (9.2.2), THIS (9.3.13)

The EQU and equal sign (=) directives can also be used to create instruction labels. If the address expression following the directive has a data type of NEAR or FAR, an instruction label is created. Please see section 5.2.3 for details.

6.5.4 Referencing Variables

Variable data types are used by the assembler to determine the size (number of bytes) of an instruction operand and to detect incompatible data type references. If necessary, the PTR operator can be used to override a variable's data type in an expression.

Example:

```
wvar    LABEL   word
dvar    DD      0
        mov     wvar,1                ; move a word value
        mov     word PTR dvar,1       ; move a word value
        pop     dvar                  ; pop a double word
        mov     wvar,ax               ; valid statement
        mov     dword PTR wvar,eax    ; valid
        mov     wvar,eax              ; error
```

Please see also: PTR (9.3.7)

6.6 Structure Definitions

386|ASM allows the programmer to define arbitrary data structures and to assign them a name. An arbitrary data structure is a collection of one or more fields, each having an assembler-defined data type. When creating variables and initializing data, it is possible to use the name of a structure in place of an assembler data declaration directive. Thus, one can create and initialize variables which are instances of arbitrary structures. Structure fields can also have default values associated with them. These values are used when an instance of a structure is declared and no other value is given at declaration time.

The remainder of this section is devoted to an explanation of the procedure for creating an arbitrary structure definition. Note that a structure definition does not itself allocate any memory. It only creates a

template for a structure and saves the default values for the fields. In order to allocate memory for a structure, use a structure declaration. Structure declarations are discussed later in this chapter.

6.6.1 STRUC Directive

Syntax:

```
name STRUC
```

The STRUC directive marks the beginning of a structure definition. The *name* parameter specifies the name of the structure to be created, which must be unique. The structure definition is terminated by the ENDS directive, and all lines between the STRUC and ENDS directives are processed as part of the structure definition. Since structure definitions do not allocate memory, they can be placed outside a segment block, if desired.

6.6.2 ENDS Directive

Syntax:

```
name ENDS
```

The ENDS directive terminates the current structure definition. The *name* parameter must be the same as the name used to start the structure definition. The ENDS directive causes the assembler to end the current structure definition and then to switch back to normal statement processing mode.

Note that the ENDS directive is also used to close an open segment (please see section 3.8). The assembler determines which form of ENDS is intended based on whether or not a structure is currently being defined.

Please see also: ENDS (3.8)

6.6.3 Defining Fields Within a Structure

Syntax:

```
{fieldname} DB value,...
{fieldname} DW value,...
{fieldname} DD value,...
{fieldname} DP value,...
{fieldname} DF value,...
{fieldname} DQ value,...
{fieldname} DT value,...
```

Once a structure definition is started, all statements preceding the terminating ENDS directive must be either comments or data declaration directives. A data declaration directive within a structure defines a structure field (with an optional name) having the data type of the directive and a size equal to the amount of memory the directive allocates. The value list which follows the directive is saved as a default value for the field when an instance of the structure is declared.

Thus a typical structure definition looks something like this:

```
name    STRUC
   ;  definitions for the structure fields go here.
name    ENDS
```

Example:

A structure which could be a node in a balanced binary tree:

```
node      STRUC

key       DB 31 DUP(?)     ; The key for the node
balance   DB ?             ; The node's balance factor
lchild    DD ?             ; A pointer to the left child
rchild    DD ?             ; A pointer to the right child

node      ENDS
```

A structure which has unnamed fields and initial values:

```
pi       STRUC

sname    DB 'PI',0       ; Structure's ID
         DB 0            ; Pad byte for alignment
```

```
value   DQ 3.1415926  ; Initial Value

pi      ENDS
```

6.7 Structure Declarations

As mentioned above, a structure definition creates only a template for data. In order to allocate and initialize memory for a structure, use a structure declaration. This section discusses how to allocate and initialize one or more instances of a structure and how to create structure variables.

6.7.1 Using a Structure Name to Allocate Memory

Syntax:

```
{name} structurename <{value,...}>
```

Once a structure has been defined, its name can be used in a similar manner to the data declaration directives for assembler–defined data types. Using a structure name in this manner causes the assembler to allocate space for an instance of the structure in the currently open segment.

The *name* parameter is optional and, if present, it creates a variable with a symbol type of structure and an address attribute of the current value of the location counter. The *structurename* parameter is the name of a structure previously defined with the STRUC directive. The angle brackets which follow the *structurename* parameter must be present and are used to contain an optional list of values used in place of the structure's default field values or initializers. One value per field is permitted in the structure being initialized. Multiple values must be separated with commas. If an initializer for a field is omitted, the default field value is used.

6.7.2 Initializers

386 | ASM restricts the types of structure fields which can be initialized. Only structure fields which are a single value or a constant string can have their default values overridden by an initializer. Any field which contains a list of values, or multiple values allocated with DUP, cannot be

initialized when an instance of a structure is allocated. A constant string's value can be replaced by any number of bytes up to its length. If the initializer is smaller than the string it is replacing, the trailing bytes of the field are padded out with spaces (20h).

6.7.3 Creating Multiple Structures with the DUP Operator

Syntax:

```
{name} structurename count DUP(<{value,...}>)
```

It is possible to use the DUP operator to create multiple instances of a structure with each one using the same set of initial values. The syntax of such a declaration is similar to that used for the data declaration directives. The *count* parameter tells 386 | ASM how many instances of the structure to create, and the optional initializer list appears inside the parentheses which follow the DUP operator. The allowable values for the initializer list are the same as if the DUP operator were not used.

Example:

Some examples of structure definitions and declarations appear below:

```
node      STRUC

key       DB 31 DUP(?)     ; Cannot be initialized
balance   DB 0             ; Can be initialized
lchild    DD ?             ; Can be initialized
rchild    DD ?             ; Can be initialized

node      ENDS

; create a node using the structure's default values
node1 node <>

; create a node and set the balance factor to 0.
node2 node <,0>

; create a node and set the balance factor to 0 and
; the child pointers to two other nodes.
node3 node <,0,node1,node2>

; create 20 nodes with a balance factor of 0 and
; NULL child pointers.
nodearray node 20 DUP(<,0,0,0>)
```

```
pi       STRUC

name     DB 'PISTRUCT'    ; Can be initialized
         DB 0             ; Can be initialized
value    DQ 3.1415926     ; Can be initialized
         DB 1,2,3,4       ; Cannot be initialized

pi       ENDS


; Create an instance using the default field values.
; Do not create a variable.
          pi <>

; Create an instance and override the structure's name
; and value fields.
newpi     pi <'NEWPI',,3.1748>

; Create an array of 10 instances of the structure
; with no name and all fields zero except the last one
; which uses the default.
piarray   pi 10 DUP(<' ',0,0.0>)
```

Please see also: DUP (6.3), Undefined Storage Operand (?) (6.4), STRUC (6.6.1)

6.8 Referencing Structures

Structure variables carry an address (segment and offset) attribute, but do not have a data type. Individual fields in a structure are most easily referenced with the structure field operator (please see section 9.3.9). The data type of the expression is obtained from the field data type. The structure field operator adds the field offset within the structure to the structure address to obtain the correct address of the structure field.

Example:

```
farp      STRUC
offs      DD        ?          ; offset in segment
segsel    DW        ?          ; segment selector
farp      ENDS
dseg      SEGMENT   word public
```

```
ptrarray farp      10 DUP (<0,dseg>)
dseg     ENDS

cseg     SEGMENT  word public

         mov      ax,ptrarray.segsel    ; load 1st ptr
         mov      es,ax                 ; in array into
         mov      edi,ptrarray.offs     ; ES:EDI

         mov      ebx,OFFSET ptrarray   ; load ptr to 2nd
         add      ebx,SIZE farp         ; struc in array
         mov      ax,[ebx].segsel       ; load 2nd ptr
         mov      fs,ax                 ; in array into
         mov      esi,[ebx].offs        ; FS:ESI

cseg     ENDS
```

Please see also: SIZE (9.3.4), Structure Field Operator (9.3.9)

6.9 RECORD Directive

Syntax:

name RECORD *fname:width{=expr}*,...

The RECORD directive allows the programmer to break up an 8-, 16-, or 32 bit value into named sub-fields, each one having an arbitrary width in bits. It is also possible to give each bit field an initial value to be used as a default when an instance of the record is created.

The *name* parameter assigns a name to the record being defined, which must be unique. Each field definition must have its field name and field width separated by a colon. If the optional default field value is specified, it must immediately follow the field width and be preceded by an equal sign. Multiple field definitions must be separated with commas.

Default field values are specified by an expression which must evaluate to a constant value small enough to fit in the specified field width. If no default value is specified, the field is given a default value of zero.

Fields in a record are allocated left to right, with the first field following the directive using the leftmost bits in the record. Succeeding fields are also allocated left to right in the remaining bits of the record.

386|ASM processes record definitions using the following logic:

1. The total size in bits of all the fields in the record is calculated, and a size for the record is chosen. A record can be 8, 16, or 32 bits wide (32 bit wide records are permitted only if the target CPU is an 80386). The assembler chooses the smallest size in which the entire record fits.

2. Next, the bit offsets for the fields are chosen. As mentioned above, the fields in a record are allocated left to right in the order in which they follow the RECORD directive. If the total size of the fields does not completely fill the record, the fields are shifted to the right to guarantee that the last bit of the last field occupies bit zero of the record. Unused high order bits are set to zero.

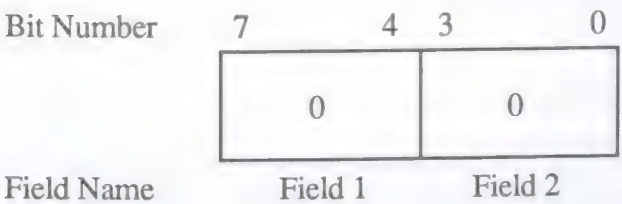
Note that a record definition does not itself allocate any memory. It only creates a template for a record and saves the default values for the fields. In order to allocate memory for a record, use a record declaration. Record declarations are discussed in the next section. Since record definitions do not allocate memory, they can be placed outside a segment block, if desired.

Example:

Each example below shows a RECORD directive and how the fields of the record it creates are organized in memory. The default values for the fields are also shown in the memory layout.

```
; An eight bit record with two four bit fields.

rec1  RECORD field1:4,field2:4
```



```
; A 16 bit record which only uses nine bits.  The
; middle field has a default value of three.

rec2  RECORD hibits:3,midbits:3=3,lobits:3
```

Bit Number	15	8 7		0
	0	000	011	000
Field Name		hibits	midbits	lobits

```
; A 32 bit record which is made up of three one byte
; fields having default values of A, B, and C.

rec3    RECORD ch1:8='A',ch2:8='B',ch3:8='C'
```

Bit Number	31 24	23 16	15 8	7 0
	0	'A'	'B'	'C'
Field Name		ch1	ch2	ch3

6.10 Record Declarations

Like a structure definition, a record definition creates only a template for data. In order to allocate and initialize memory for a record, you must use a record declaration. This section discusses how to allocate and initialize one or more instances of a record and how to create record variables.

6.10.1 Using a Record Name to Allocate Memory

Syntax:

```
{name} recordname <{value,...}>
```

Once a record is defined, its name can be used in a manner similar to the data declaration directives for assembler-defined data types. Using a record name in this manner causes the assembler to allocate space for an instance of the record in the currently open segment. The amount of

space allocated is one, two, or four bytes, depending on the total size of the fields in the record.

The *name* parameter is optional and, if present, it creates a variable with a symbol type of record and an address attribute of the current value of the location counter. In addition, a record variable can be referenced as a variable of type BYTE, WORD, or DWORD, depending on whether the record size is one, two, or four bytes. The *recordname* parameter is the name of a record defined with the RECORD directive. The angle brackets which follow the *recordname* parameter must be present and are used to contain an optional list of values to be used in place of the record's default field values. One value is permitted per field in the record being initialized. Multiple values must be separated with commas. If an initializer for a field is omitted, the default value given at definition time is used.

An initializer for a record field must be an absolute value and must fit in the width of the field it is initializing. If the initializer is relocatable or too large for the field, an error is posted and the default value for the field is used instead.

6.10.2 Creating Multiple Records with the DUP Operator

Syntax:

```
{name} recordname count DUP(<{value,...}>)
```

It is possible to use the DUP operator to create multiple instances of a record, with each having the same set of initial values. The syntax of such a declaration is similar to that used for the data declaration directives. The *count* parameter tells 386|ASM how many instances of the record to create, and the optional initializer list appears inside the parentheses which follow the DUP operator. The allowable values for the initializer list are the same as if the DUP operator were not used.

Example:

```
rec1  RECORD field1:4,field2:4

; a one byte record using default values (in this
; case, zero since no defaults were given).
```

```
byte1 rec1 <>

; a one byte record with the low nibble equal to 3.
byte2 rec1 <,3>

; a one byte record with value 12h
byte4  rec1 <1,2>

; ten one byte records with value 34h
bytearray rec1 10 DUP(<3,4>)

rec3    RECORD ch1:8='A',ch2:8='B',ch3:8='C'

; a 32 bit record with value 00000043h.
dword1 rec3 <0,0>

; a 32 bit record with value 00123456h
dword2 rec3 <10h+2,4*12+4,56h>
```

Please see also: DUP (6.3), Undefined Storage Operand (?) (6.4), RECORD (6.9)

6.11 Referencing Records

Record variables carry an address (segment and offset) attribute, and can also be referenced as BYTE, WORD, or DWORD variables, depending on whether they are 8, 16, or 32 bits wide. The WIDTH operator can be used to obtain the width, in bits, of a record or record field. The MASK operator can be used to obtain a bit mask for the defined bits in the record, or for a single field in the record. The SIZE operator gives the size of the record in bytes (one, two, or four bytes).

Example:

```
flags      RECORD    inuse:1,type:4
farray     flags     10 DUP (<0,0>)
flag2      PROC      near
           mov       al,farray+SIZE flags     ; load 2nd
                                              ; flag byte
           test      al,MASK inuse            ; call type
           jz        #not_used                ; check routine
           call      type_chk                 ; if in use
```

```
#not_used:
            ret
flag2    ENDP
```

Please see also: WIDTH (9.3.5), MASK (9.3.5), SIZE (9.3.4)

Chapter 7

Global Symbol Definitions

7.1 Introduction

It is often necessary to break up a large program into several modules which are assembled separately. A problem which occurs when modules are separately assembled is that subroutines and data once accessible to the entire program are now accessible only inside the module in which they are defined. The global symbol management directives provide a solution to this problem by allowing variables, instruction labels, and absolute constants from one module to be referenced in the other modules of a program.

The PUBLIC directive is used to make available the value of a symbol in the module being assembled to other modules in a program. The EXTRN directive is used to reference the value of a symbol which is defined in another module. The syntax and operation of these directives is described in more detail below.

When the assembler encounters a reference to an external symbol, it gives the symbol a value of zero in its expression and records the fact that the symbol was referenced at that location in the file. When the object files for the modules are combined, 386 | LINK adds the actual value of the external symbol to the expression in which it was used.

7.2 PUBLIC Directive

Syntax:

```
PUBLIC name,...
```

The PUBLIC directive makes available the value of one or more symbols defined in the module being assembled to other modules in the program.

The *name* parameter can be a variable, an instruction label or an absolute constant.

Defining a symbol as public does not affect its use in its defining module. In its module, it is treated just like any other local symbol. The only difference is that some extra information about the symbol is recorded in the assembler object output file which enables 386|LINK to resolve references to it.

Exactly one PUBLIC definition must appear for a global symbol. If a symbol is defined as public in more than one module of a program, 386|LINK signals a duplicate symbol definition.

Example:

The example below shows how the values of several different symbols are made available to other modules in a program.

```
PUBLIC c1, c2, v1, v2, l1, l2, l3

c1 EQU 1000h
c2 = 'hi'
v1     DW 10FFh
v2     LABEL word

l1:
l2     LABEL near
l3     PROC far
```

Please see also: EXTRN (7.3)

7.3 EXTRN Directive

Syntax:

```
EXTRN name:type,...
```

The EXTRN directive tells the assembler that the symbol *name* is defined outside the module being assembled and has a data type of *type*. The type specifier can be any of the assembler-defined data types, the name of a structure or record definition, or the keyword ABS, which indicates that the symbol is an absolute constant. A list of valid type specifiers follows:

BYTE
WORD
DWORD
PWORD / FWORD
QWORD
TBYTE
NEAR
FAR
ABS
structurename
recordname

Although the value of the symbol is not known until link time, the assembler assumes a segment for variables and labels based on the position of their defining EXTRN statement. If the EXTRN directive appears inside a segment block, all variables and labels defined by it are assumed to be in that segment, and the assembler generates override bytes as necessary. If the directive does not appear inside a segment block, labels and variables defined by it are not assumed to be in any particular segment. In this case, the assembler assumes that no override bytes will be necessary to reference them. In any case, the segment to which any expression is relative can be specified with the segment override operator, if necessary.

Example:

The following example demonstrates how a variable, a label, and a constant external to the current module can be referenced:

```
EXTRN number:WORD, scale:ABS, print_num:NEAR

foo     PROC near

        mov     ax,number          ; Get the number.
        add     ax,scale           ; Scale it.
        call    print_num          ; And print its value.
        ret                        ;

foo     ENDP
```

Please see also: PUBLIC (7.2), Segment Overrides (9.3.10)

7.4 Redefining an External Symbol as Public

386|ASM permits a symbol defined as external to be redefined as a public symbol within the module being assembled with one restriction: the symbol cannot be referenced between the line which declares it external and the line which changes its definition to public. That is to say, once an external is referenced in a module, its definition is locked as external and cannot be changed. In order to avoid assembler errors, the following sequence must be used to correctly redefine an external as public:

```
EXTRN foo:near      ; External Definition.
   .
   .       ; foo cannot be referenced here.
   .
PUBLIC foo        ; Redefinition as a public.
   .
   .       ; foo CAN be referenced here, and will follow
   .       ; the normal rules for forward references
   .       ; (the data type specified in the EXTRN is
   .       ; removed by the PUBLIC redefinition).
   .
foo    PROC NEAR  ; Actual declaration of foo.

                  ; foo can be referenced here.
```

Any other ordering of a symbol's EXTRN declaration, its PUBLIC redefinition, and references to it are considered to be an error.

This feature makes the definition of interfaces in a large system less complicated. The global variables and entry points for all modules in the system are defined as external in one header file which is included in all modules. Each module includes this external definitions file, then immediately follows with PUBLIC definitions for any symbols which are actually defined in the module. Thus, each module does not have to explicitly declare which external symbols it is going to reference, and all modules include one header file to define all the globals in the system.

Chapter 8

Assembler Control Directives

8.1 Introduction

The assembler control directives are used to change parameters which are global to the entire assembly. This chapter includes descriptions of the instruction set control directives, the listing file control directives, the conditional directives, and miscellaneous control directives.

8.2 Instruction Set Directives

The instruction set directives select the target CPU and the numeric coprocessor for the assembly. Some assembler directives and instruction mnemonics are target CPU specific and are enabled/disabled by these directives. The instruction set directives which appear in a source file override any values specified on the command line. They are, however, required to appear before the first segment of the module is defined.

Please see also: Instruction Set Switches (2.2.5)

8.2.1 .8086 Directive

Syntax:

```
.8086
```

The .8086 directive specifies the target CPU for the assembly as an 8086 or an 8088. All instructions specific to the 80186, 80188, 80286, or 80386 are disabled, and any segments defined in the module use a 16 bit location counter. After processing this directive, 386 | ASM essentially becomes an 8086 assembler.

8.2.2 .186 Directive

Syntax:

```
.186
```

The .186 directive specifies the target CPU for the assembly as an 80186 or an 80188. All instructions specific to the 80286 or the 80386 are disabled, and any segments defined in the module use a 16 bit location counter.

8.2.3 .286 and .286c Directives

Syntax:

```
.286
.286c
```

The .286 and .286c directives are synonymous and can be used interchangeably. They select an 80286 as the target CPU for the assembly, but do not enable the protected instructions. Any instructions specific to the 80386 are disabled, and any segments defined in the module use a 16 bit location counter.

8.2.4 .286p Directive

Syntax:

```
.286p
```

The .286p directive specifies the target CPU for the assembly as an 80286 and also enables the protected instructions. All instructions specific to the 80386 are disabled, and any segments defined in the module use a 16 bit location counter.

The .286p directive is equivalent to the .286 directive followed by the .PROT directive.

8.2.5 .386 and .386c Directives

Syntax:

```
.386
.386c
```

The .386 and .386c directives are synonymous and can be used interchangeably. They select the target CPU for the assembly as an 80386, but do not enable the protected instructions. When the target CPU is an 80386, the default use type for segments is USE32, and a 32 bit location counter is used.

The 80386 nonprotected instruction set is the default instruction set enabled by 386 | ASM. If the source file does not have an instruction set directive and if no instruction set switch is used on the command line, the nonprotected 80386 instruction set is enabled.

8.2.6 .386p Directive

Syntax:

```
.386p
```

The .386p directive specifies the target CPU for the assembly as an 80386 and also enables the protected instructions. The default use type for segments is USE32, and a 32 bit location counter is used.

The .386p directive is equivalent to the .386 directive followed by the .PROT directive.

8.2.7 .PROT Directive

Syntax:

```
.PROT
```

The .PROT directive enables the protected instructions without changing the target CPU for the assembly. Though it only makes sense to use this directive when the target CPU is an 80286 or an 80386, it is processed with no effect if the target CPU is an 8086 or an 80186.

8.2.8 .8087 Directive

Syntax:

```
.8087
```

The .8087 directive enables the 8087 floating point coprocessor instruction set. It does not change the target CPU for the assembly.

8.2.9 .287 Directive

Syntax:

```
.287
```

The .287 directive enables the 80287 floating point coprocessor instruction set. It does not change the target CPU for the assembly. One difference between the 8087 and the 80287 instruction sets is the addition of the FSETPM instruction for the 80287. Also, the FENI and FDISI instructions generate no object code if the target CPU is an 80287.

The important change which takes place when the 80287 is selected is that 386|ASM does not generate FWAIT instructions preceding floating point instructions if the target CPU is an 80286 or an 80386 and the target coprocessor is an 80287. This is because floating point operations are synchronized by the processors and the FWAIT instructions are unnecessary.

8.2.10 .387 Directive

Syntax:

```
.387
```

The .387 directive enables the 80387 floating point coprocessor instruction set. It does not change the target CPU for the assembly.

8.3 List File Control Directives

The list file control directives are used to control the format of the assembler listing file. 386 | ASM provides directives which set the page title and subtitle, and directives to control which source lines are copied to the listing file.

8.3.1 TITLE Directive

Syntax:

```
TITLE text
```

The TITLE directive sets the title field of the listing file page header to the *text* which follows the directive on the line. Only one title is permitted per module. If 386 | ASM encounters more than one title for a module, it signals an error, and the first title is used.

If the module does not have a name specified with the NAME directive, the first six characters of the module's title are used as its name.

Example:

```
;
; Set the title of the module to "386|ASM TITLE
; demonstration" and the name of the module
; to "386|AS".
;

TITLE 386|ASM TITLE demonstration
```

Please see also: SUBTTL (8.3.2), NAME (3.3)

8.3.2 SUBTTL Directive

Syntax:

```
SUBTTL {text}
```

The SUBTTL directive is used to set the subtitle field of the listing file page header to the *text* which follows the directive on the line. The *text*

parameter is optional; if no text follows the directive, the subtitle field of the header is cleared.

More than one SUBTTL directive may appear in a module being assembled. Each one will take effect on the next page of the listing file.

Example:

```
;
; Set the subtitle field to Subtitle #1.
;

SUBTTL Subtitle #1
```

Please see also: TITLE (8.3.1)

8.3.3 PAGE Directive

Syntax:

```
PAGE {length},{width}
PAGE +
PAGE
```

The PAGE directive has different effects, depending on the syntax in which it is used. If used with an optional *length* and/or *width*, it will change the listing file page length and width to the values specified. The *length* parameter, if present, must be between 10 and 255 lines, and the *width* parameter must be between 60 and 132 characters. If the length or width is omitted, its value remains unchanged.

If the directive is followed by a single plus sign (+), it causes 386|ASM to increment the section number field of the page number used in the page header, and generates a page break in the listing file.

If the directive is used with no arguments, it generates a page break in the listing file.

Example:

```
; Set the page length and width to 60 lines and
; 132 columns, respectively.
PAGE 60,132
```

```
; Set the length to 60 without changing the width.
PAGE 60,

; Set the width without changing the length.
PAGE ,132

; Bump the section number of the page number.
PAGE +

; Generate a page break in the listing file.
PAGE
```

8.3.4 .LIST Directive

Syntax:

```
.LIST
```

The .LIST directive enables the listing of program statements in the listing file. It is used to reverse the effect of the .XLIST directive. Listing of program statements is enabled by default when assembly begins.

8.3.5 .XLIST Directive

Syntax:

```
.XLIST
```

The .XLIST directive disables the listing of program statements in the listing file. All subsequent source lines are not displayed in the listing file until a .LIST directive is processed.

8.3.6 .LISTI Directive

Syntax:

```
.LISTI
```

The .LISTI directive causes program statements included from another file to be listed in the listing file. It is used to reverse the effect of the .XLISTI directive. This is the default when assembly begins.

Please see also: INCLUDE (3.5)

8.3.7 The .XLISTI Directive

Syntax:

```
.XLISTI
```

The .XLISTI directive causes program statements included from another file to be suppressed from the listing file. All subsequent source lines included using the INCLUDE directive are not listed in the listing file until a .LISTI directive is processed.

Please see also: INCLUDE (3.5)

8.3.8 .LFCOND Directive

Syntax:

```
.LFCOND
```

The .LFCOND directive causes text in false conditional blocks to be displayed in the assembler listing file even though it is not being assembled.

Please see also: Conditional Assembly (8.4)

8.3.9 .SFCOND Directive

Syntax:

```
.SFCOND
```

The .SFCOND directive suppresses the listing of all subsequent false conditional blocks in the listing file. This is the default when assembly begins.

Please see also: Conditional Assembly (8.4)

8.3.10 .TFCOND Directive

Syntax:

```
.TFCOND
```

The .TFCOND directive toggles the current state of the "list false conditional blocks" flag in the assembler. If false conditional blocks are currently being listed, they will be suppressed. If they are being suppressed, the .TFCOND directive causes them to be listed again.

Please see also: Conditional Assembly (8.4)

8.3.11 .LALL Directive

Syntax:

```
.LALL
```

The .LALL directive causes all program statements generated due to macro expansions to be listed in the listing file.

Please see also: Macro Expansions (10.3)

8.3.12 .SALL Directive

Syntax:

```
.SALL
```

The .SALL directive causes any program statements generated due to macro expansions to be suppressed from the listing file.

Please see also: Macro Expansions (10.3)

8.3.13 .XALL Directive

Syntax:

```
.XALL
```

The .XALL directive causes only those program statements from macro expansions which generate object code to be listed in the listing file. This is the default when assembly begins.

Please see also: Macro Expansions (10.3)

8.4 Conditional Assembly Directives

The conditional assembly directives allow the programmer to specify whether or not a certain block of instructions will be assembled based on a conditional parameter. The desired condition is specified by selecting the appropriate conditional assembly directive, described in sections 8.4.1–8.4.8.

The simplest form of conditional assembly is a code block which begins with a conditional assembly directive, and is terminated by the ENDIF directive. The instructions in the conditional block are assembled only if the specified condition is met. The general structure of this form of conditional assembly looks like this:

```
IF       (some condition is true)
.
.        (instructions to be assembled if the
.         condition is met)
.
ENDIF    (terminate the conditional block)
```

Conditional assembly directives can be used with the ELSE directive to select one of two blocks of code to be assembled. The first block of code, referred to as the "true portion" of the conditional block, begins with the conditional assembly directive and is terminated by the ELSE directive. The true portion of the conditional block is assembled only if the specified condition is met. The second block of code, referred to as the "false portion" of the conditional block, begins with the ELSE directive and is terminated by the ENDIF directive. The false portion of the conditional block is assembled only if the specified condition is not met. The general structure of this form of conditional assembly looks like this:

```
IF      (some condition is true)
.
.
.       (true portion: instructions to be
.        assembled if the condition is met)
.
ELSE
.
.       (false portion: instructions to be
.        assembled if the condition is not met)
.
ENDIF   (terminate the conditional block)
```

Conditional blocks may be nested up to 255 levels. An ENDIF is always matched to the nearest preceding IF or ELSE.

8.4.1 IF Directive

Syntax:

`IF` *`expression`*

The IF directive assembles the instructions in the true portion of a conditional block, if the *expression* which follows it on the line evaluates to a non-zero value.

Example:

```
IF      5
DB      0   ; will be assembled
ELSE
DB      1   ; will not be assembled
ENDIF
```

8.4.2 IFE Directive

Syntax:

`IFE` *`expression`*

The IFE directive assembles the instructions in the true portion of a conditional block, if the *expression* which follows it on the line evaluates to zero.

Example:

```
IFE     5
DB      0    ; will not be assembled
ENDIF
```

8.4.3 IFDEF Directive

Syntax:

IFDEF *name*

The IFDEF directive assembles the instructions in the true portion of a conditional block, if the *name* which follows it on the line is a user-defined symbol or an assembler keyword. The *name* parameter must not be a forward reference.

Example:

```
lab:
        IFDEF   lab
        jmp     lab      ; will be assembled
        ENDIF
```

Please see also: -DEFINE (2.2.7)

8.4.4 IFNDEF Directive

Syntax:

IFNDEF *name*

The IFNDEF directive assembles the instructions in the true portion of a conditional block, if the *name* which follows it on the line is not a user-defined symbol or an assembler keyword.

Example:

```
        IFNDEF   lab
lab     LABEL    near     ; define label only if not
                          ; already defined
        ENDIF
```

Please see also: -DEFINE (2.2.7)

8.4.5 IFB Directive

Syntax:

```
IFB <string>
```

The IFB directive assembles the instructions in the true portion of a conditional block if the *string* which is enclosed by angle brackets is blank. IFB is useful for testing the presence of macro parameters.

Example:

```
def_str  MACRO   param    ; define 0-terminated string
         IFB     <param>
         DB      0
         ELSE
         DB      '&param', 0
         ENDIF
         ENDM
```

Please see also: Macro Expansions (10.3), Parameter Substitution (10.11)

8.4.6 IFNB Directive

Syntax:

```
IFNB <string>
```

The IFNB directive assembles the instructions in the true portion of a conditional block if the *string* which is enclosed by angle brackets is not blank. IFNB is useful for testing the presence of macro parameters.

Example:

```
def_str  MACRO   param     ; define 0-terminated string
         IFNB    <param>
         DB      '&param'
         ENDIF
         DB      0
         ENDIF
```

Please see also: Macro Expansions (10.3), Parameter Substitution (10.11)

8.4.7 IFIDN Directive

Syntax:

```
IFIDN <str1,str2>
```

The IFIDN directive assembles the instructions in the true portion of a conditional block if the two strings enclosed by angle brackets are identical.

Example:

```
end_str  MACRO  flag   ; terminate string conditionally
         IFIDN  <flag>,<TRUE>
         DB     0
         ENDIF
         ENDM
```

Please see also: Macro Expansions (10.3), Parameter Substitution (10.11)

8.4.8 IFDIF Directive

Syntax:

```
IFDIF <str1,str2>
```

The IFDIF directive assembles the instructions in the true portion of a conditional block if the two strings enclosed by angle brackets are different.

Example:

```
end_str  MACRO  flag   ; terminate string conditionally
         IFDIF  <flag>,<FALSE>
         DB     0
         ENDIF
         ENDM
```

Please see also: Macro Expansions (10.3), Parameter Substitution (10.11)

8.4.9 ELSE Directive

Syntax:

```
ELSE
```

The ELSE directive terminates the true portion of a conditional block and marks the start of a block of instructions, which should be assembled if the true block is not assembled. This block of instructions is called the false portion of a conditional block and must be terminated by the ENDIF directive.

8.4.10 ENDIF Directive

Syntax:

```
ENDIF
```

The ENDIF directive terminates a conditional block. It must appear after the last statement in the true portion of the block or after the last statement of the false portion of the block, if it exists.

8.5 Conditional Error Directives

The conditional error directives are used to force an assembler error, if the given condition is met. They are useful for signaling an error if an assumption made turns out to be invalid.

8.5.1 .ERR Directive

Syntax:

```
.ERR
```

The .ERR directive forces an assembler error unconditionally.

8.5.2 .ERRE Directive

Syntax:

```
.ERRE expression
```

The .ERRE directive forces an assembler error if *expression* evaluates to zero.

8.5.3 .ERRNZ Directive

Syntax:

```
.ERRNZ expression
```

The .ERRNZ directive forces an assembler error if *expression* evaluates to a non–zero value.

8.5.4 .ERRDEF Directive

Syntax:

```
.ERRDEF name
```

The .ERRDEF directive forces an assembler error if *name* is a user–defined symbol or an assembler keyword.

8.5.5 .ERRNDEF Directive

Syntax:

```
.ERRNDEF name
```

The .ERRNDEF directive forces an assembler error if *name* is not a user-defined symbol or an assembler keyword.

8.5.6 .ERRB Directive

Syntax:

```
.ERRB <string>
```

The .ERRB directive forces an assembler error if *string* is blank.

8.5.7 .ERRNB Directive

Syntax:

```
.ERRNB <string>
```

The .ERRNB directive forces an assembler error if *string* is not blank.

8.5.8 .ERRIDN Directive

Syntax:

```
.ERRIDN <str1,str2>
```

The .ERRIDN directive forces an assembler error if strings *str1* and *str2* are identical.

8.5.9 .ERRDIF Directive

Syntax:

```
.ERRDIF <str1,str2>
```

The .ERRDIF directive forces an assembler error if strings *str1* and *str2* are different.

8.6 Miscellaneous Control Directives

8.6.1 %OUT Directives

Syntax:

```
%OUT text
%OUT1 text
%OUT2 text
```

The %OUT directives are used to copy the *text* which follows them on the line to the user's terminal. The %OUT directive copies the text on both passes of the assembly, while %OUT1 and %OUT2 only copies the text on pass one or pass 2, respectively.

8.6.2 .RADIX Directive

Syntax:

```
.RADIX expression
```

The .RADIX directive sets the default radix for all following numbers in the source code to *expression*. The *expression* is always evaluated in base 10, regardless of the current default radix. The value of *expression* must be between 2 and 16.

Chapter 9

Expressions

9.1 Introduction

Many assembly language instructions or directives require operands. An operand may be a constant value, a relocatable value, or the name of an 80386 register.

For most statements which require operands, a single value may be replaced by a more complex expression which evaluates to a constant or relocatable value. An expression combines operands through the use of expression operators to calculate a single value. For the remainder of this chapter, the word operand will refer to a single constant value or a relocatable value in an expression (as opposed to an assembly language statement operand, which may be either a single value or an expression which evaluates to a single value).

9.2 Operands

An expression operand is either a constant value, a user-defined symbol (an identifier), or an assembler reserved word. An operand is either a constant value or a relocatable value. Relocatable values are values that are modified (relocated) when the assembled object code is linked, or when the linked program is loaded into memory for execution.

9.2.1 Constant Operands

A constant operand is a number, a string constant, or a user-defined symbol with a constant value. Table 9-1 shows the types of user-defined symbols that assume constant values and the value each type assumes in an expression.

TABLE 9-1
CONSTANT SYMBOL TYPES

Symbol Type	Value
Integer constant	The value given by the EQU or = directive when the symbol was created.
Structure definition	The size, in bytes, of the structure.
Structure field	The offset, in bytes, of the field within the structure.
Record definition	The size, in bits, of the record.
Record field	The offset, in bits, of the field within the record.

Please see also: Defining Constants With the EQU and = Directives (Chapter 4), Structure Definitions (6.6), RECORD (6.9)

9.2.2 Relocatable Operands

A relocatable operand is a user-defined symbol whose final value is not determined until the object modules are linked to make a task image, or until the task image is loaded into memory for execution. Table 9-2 shows relocatable user-defined symbols and their values.

TABLE 9-2
RELOCATABLE OPERANDS

Symbol Type	Value
Variable, structure, record, label, or procedure label	The offset of the symbol within the segment in which it is defined. The final value is assigned at link time.
$	A special symbol which represents the current offset within the current segment. It has the same attributes as a near label. The final value is assigned at link time.
Segment or group	For 8086, 8088, 80186 or 80188 programs, or for 80286 or 80386 real mode programs, this is the paragraph address (where paragraphs begin on 16-byte boundaries) at which the segment or group is loaded into memory. For 80386 or 80286 protected mode programs, this is the index of a segment descriptor in a table maintained by the operating system. The segment descriptor gives the segment location in memory and other information about the segment. In either case, the final value is assigned at the time the program is loaded into memory for execution.

Please see also: Variables and Data Declarations (Chapter 6), SEGMENT (3.7), GROUP (3.9), Chapter 11

9.2.3 Assembler Reserved Words

A few assembler reserved words are valid expression operands. Assembler reserved words that are not valid operands are simply assigned a constant value of zero when encountered in an expression. Table 9-3 shows assembler reserved words which are valid expression operands, and the values they are assigned.

TABLE 9-3
RESERVED WORD VALUES

Reserved Word	Value
Register names	Register names are valid in effective address expressions (see section 9.4) used with instruction opcodes.
`BYTE`	A constant value of one
`WORD`	A constant value of two
`DWORD`	A constant value of four
`PWORD`	A constant value of six
`QWORD`	A constant value of eight
`TBYTE`	A constant value of ten
`NEAR`	A constant value of -1
`FAR`	A constant value of -2

9.2.4 Forward References

An expression which uses an identifier (a user–defined symbol) before it is defined in the source code file is said to be a forward reference to the identifier. Forward references are permitted, provided the assumptions made by 386 | ASM about the identifier cause at least as much object code to be generated on pass one of the assembly as is written to the object file on pass 2.

If 386 | ASM generates an illegal forward reference error, the condition may be corrected either by moving the identifier definition in front of the statement that references it, or by giving the assembler more information about the identifier in the statement that forward references it. The PTR operator and the segment override operator (:) are used to give the assembler additional information about forward referenced identifiers.

In some cases, it may not be possible to give 386 | ASM sufficient information about the forward reference. Then, the only option is to

move the definition above the statement that references it. For example, the text substitution symbol defined by the statement

```
long_str EQU 'This is a long string'
```

cannot be forward referenced under any circumstances by the statement

```
DB long_str
```

because the assembler needs to know how much space to reserve for the string on pass one.

Please see also: PTR (9.3.7), Segment Override (9.3.10), SHORT (9.3.12)

9.3 Operators

Expression operators are used to combine multiple operands into a single constant value or a relocatable value. Operators are assigned a precedence ranking which is used to determine the order of evaluating expressions. Operators of higher precedence are evaluated before operators of lower precedence. Operators of equal precedence are evaluated left to right. Operator precedence may be overridden through the use of parentheses within the expression. Operations within parentheses are always evaluated before adjacent operations. Appendix H contains tables which summarize all the expression operators and show operator precedence.

9.3.1 Arithmetic Operators

The arithmetic operators perform the standard mathematic operations. Table 9-4 shows the arithmetic operators, their syntax, and their use.

TABLE 9-4
ARITHMETIC OPERATORS

Operator	Syntax	Use
+	*expr1* + *expr2*	Adds the two operands. If both operands are constant, the result is constant. If one operand is relocatable, the result is relocatable. At least one operand must be constant.
-	*expr1* - *expr2*	Subtracts *expr2* from *expr1*. If both operands are constant, the result is constant. If both operands are relocatable relative to the same segment, the result is a constant. If the left hand operand is relocatable and the right hand operand is constant, the result is relocatable. Any other use of relocatable operands is illegal.
*	*expr1* * *expr2*	Multiplies the two operands. Both operands must be constant, and the result is constant.
/	*expr1* / *expr2*	Divides *expr1* by *expr2*, truncating the fractional part of the result. Both operands must be constant, and the result is constant.
MOD	*expr1* MOD *expr2*	The modulo operator. Gives the remainder when *expr1* is divided by *expr2*. Both operands must be constant, and the result is constant.
unary +	+ *expr*	The unary addition operator. Yields the value of *expr*. The operand may be either constant or relocatable, and the result is the same.
unary -	- *expr*	The unary minus operator. Inverts the sign of *expr*. The operand must be constant, and the result is constant.

Example:

```
var1  DB     0
var2  DB     0
      DW     2 + 3          ; constant value 5
      DW     2 + var1       ; relocatable value
      DD     var2 + 3       ; relocatable value
```

```
DB      2 - 3          ; constant value -1
DW      var2 - var1    ; constant value 1
DW      var2 - 1       ; relocatable value
DD      2 * 3          ; constant value 6
DD      9 / 2          ; constant value 4
DD      9 MOD 2        ; constant value 1
DB      +1             ; constant value 1
DB      -1             ; constant value -1
DB      +(2 - 3)       ; constant value -1
DB      -(2 - 3)       ; constant value 1
```

9.3.2 Bitwise Operators

The bitwise operators are provided for bit manipulation of numeric values. All operands to the bitwise operators must be constant, and the result is always constant. Table 9-5 shows the bitwise operators.

TABLE 9-5
BITWISE OPERATORS

Operator	Syntax	Use
NOT	NOT *expr*	Result is the one's complement of *expr*. When the NOT operator is the last operator applied in an expression, less stringent rules are applied for posting an integer overflow error. This is so using NOT to turn off the most significant bit of a flag word or byte will not cause an overflow error.
AND	*expr1* AND *expr2*	Result is the bitwise AND of the two operands; that is, all bits that are one in both operands are also one in the result, and all bits that are zero in either operand are 0 in the result.
OR	*expr1* OR *expr2*	Result is the bitwise inclusive OR of the two operands; that is, all bits that are one in either operand are one in the result, and all bits that are zero in both operands are 0 in the result.

(cont.)

TABLE 9-5, CONTINUED

Operator	Syntax	Use
XOR	*expr1* XOR *expr2*	Result is the bitwise exclusive OR of the two operands; that is, all bits that are the same in the two operands are zero in the result, and all bits that are different in the two operands are one in the result.
SHL	*expr1* SHL *expr2*	Result is *expr1* shifted left by the number of bits specified by the value of *expr2*. Zero bits are shifted in on the right.
SHR	*expr1* SHR *expr2*	Result is *expr1* shifted right by the number of bits specified by the value of *expr2*. Zero bits are shifted in on the left.
LOW	LOW *expr*	Result is byte zero of *expr*. Equivalent to *expr* AND 0FFh.
HIGH	HIGH *expr*	Result is byte one of *expr*. Equivalent to (*expr* AND 0FF00h) SHR 8.
LOWW	LOWW *expr*	Result is bytes zero and one of *expr*. Equivalent to *expr* AND 0FFFFh.
HIGHW	HIGHW *expr*	Result is bytes two and three of *expr*. Equivalent to (*expr* AND 0FFFF0000h) SHR 16

Example:

```
DB      NOT 0Fh                 ; 0F0h
DB      0FFh AND 0Fh            ; 0Fh
DB      0F0h OR 0Fh             ; 0FFh
DB      0FFh XOR 0Fh            ; 0F0h
DB      0Fh SHL 2               ; 03Ch
DB      0FFh SHR 3              ; 1Fh
DB      0Fh SHL (1 + 3)         ; 0F0h
DD      LOW 0FFA5A5FFh          ; 000000FFh
DD      HIGH 0FFA5A5FFh         ; 000000A5h
DD      LOWW 0FFA5A5FFh         ; 0000A5FFh
DD      HIGHW 0FFA5A5FFh        ; 0000FFA5h
```

9.3.3 Relational Operators

The relational operators compare two operands and return a TRUE or FALSE value as their result. Both operands must be constant values. A TRUE result is returned as negative one (all bits set to one), and a FALSE result is returned as zero. A summary of the relational operators appears in Table 9-6.

TABLE 9-6
RELATIONAL OPERATORS

Operator	Syntax	Use
EQ	*expr1* EQ *expr2*	Result is TRUE if the operands are equal.
NE	*expr1* NE *expr2*	Result is TRUE if the operands are not equal.
LT	*expr1* LT *expr2*	Result is TRUE if *expr1* is less than *expr2*.
GT	*expr1* GT *expr2*	Result is TRUE if *expr1* is greater than *expr2*.
LE	*expr1* LE *expr2*	Result is TRUE if *expr1* is less than or equal to *expr2*.
GE	*expr1* GE *expr2*	Result is TRUE if *expr1* is greater than or equal to *expr2*.

Example:

```
DW      2 EQ 3          ; 0
DW      2 NE 3          ; 0FFFFh
DB      2 LT 3          ; 0FFh
DW      2 GT 3          ; 0
DD      3 LE 3          ; 0FFFFFFFFh
DW      2 GE 3          ; 0
```

9.3.4 LENGTH and SIZE Operators

Syntax:

```
LENGTH variable
SIZE variable|structure|record
```

The LENGTH operator returns the number of elements of storage allocated for a variable. The size of each element depends on the data type of the variable and is not related to the value returned by LENGTH.

The SIZE operator returns the number of bytes of storage allocated for a variable. It is equivalent to (LENGTH *variable*) * (TYPE *variable*).

The SIZE operator can be used with structure and record variables, and with structure and record definition names, to obtain the size in bytes of the structure or record. The LENGTH operator cannot be used with structures or records.

Example:

```
s1       STRUC
         DB       0
         DW       0
s1       ENDS
r1       RECORD   f1:8,f2:4
struc1   s1       <>
rec1     r1       <>

var1     DB       0
var2     DB       0,1,2
var3     DW       0,1,2
var4     DD       0,10 DUP (?),1

         DW       LENGTH var1     ; 1
         DW       LENGTH var2     ; 3
         DW       LENGTH var3     ; 3
         DW       LENGTH var4     ; 12

         DW       SIZE var1       ; 1
         DW       SIZE var2       ; 3
         DW       SIZE var3       ; 6
         DW       SIZE var4       ; 48

         DW       SIZE s1         ; 3
         DW       SIZE struc1     ; 3
         DW       SIZE r1         ; 2
         DW       SIZE rec1       ; 2
```

Please see also: Variables and Data Declarations (Chapter 6)

9.3.5 WIDTH and MASK Operators

Syntax:

```
WIDTH recdefname|recfldname
MASK recdefname|recfldname
```

The WIDTH operator returns the number of defined bits in a record definition or the number of bits in a record field. (Note that the WIDTH of a record definition is not equal to the amount of storage that is allocated when the record definition is used, unless all of the bits in the record definition are defined. The size of a record is always 8, 16, or 32 bits, and may be obtained by using the record name directly, without any expression operators.)

The MASK operator returns a mask for the defined bits in a record definition or a mask for a single field in its record.

Example:

```
rdef1 RECORD  rfld1:2,rfld2:8,rfld3:1

rec1  rdef1  <1,5,0>      ; Allocate storage for a record

      DW      WIDTH rdef1  ; 11
      DW      WIDTH rfld1  ; 2
      DW      WIDTH rfld2  ; 8

      DW      MASK rdef1   ; 07FFh
      DW      MASK rfld1   ; 0600h
      DW      MASK rfld2   ; 01FEh
      DW      MASK rfld3   ; 0001h

      DW      rdef1        ; 16 - size of record, in bits
      DW      rfld1        ; 9 - offset of field, in bits
      DW      SIZE rdef1   ; 2 - size of record, in bytes
      DW      SIZE rec1    ; 2 - size of record, in bytes
```

Please see also: RECORD (6.9)

9.3.6 TYPE Operator

Syntax:

```
TYPE expression
```

The TYPE operator returns a constant value representing the data type of *expression*. For variable data types, the value returned is the size of the data type in bytes. For label data types, the value returned is negative one if the label is NEAR, and negative two if the label is FAR. Please see Table 9-3 for a complete list of values returned.

For convenience, the TYPE operator can be used as synonym for the SIZE operator for structures and records. It returns the size in bytes of the structure or record.

Example:

```
var1   DB       0
var2   DD       0
lab1   LABEL    near
       DW       TYPE var1     ; 1
       DW       TYPE var2     ; 4
       DW       TYPE lab1     ; -1
```

Please see also: Variables and Data Declarations (Chapter 6), Instruction Labels, Control Transfer, and Procedure Blocks (Chapter 5)

9.3.7 PTR Operator

Syntax:

```
type PTR expression
```

The PTR operator gives *expression* a data type of *type*, where *type* is either one of the data type reserved words or their integer value equivalents given in Table 9–3. The PTR operator is useful for giving the assembler information about the data type of forward referenced symbols. It causes the assembler to assume that a forward referenced symbol has a data type which is different from the default assumed for a forward referenced symbol.

Example:

```
var1   DB      0
var2   DQ      0
          mov    al,byte PTR var2              ; load low byte
                                               ; of var2
          mov    ah,(TYPE var1) PTR var2 + 1  ; load second
                                               ; byte of var2
          call   far PTR lab1                  ; tell assembler
                                               ; that forward
                                               ; referenced label
                                               ; is of type FAR
lab1   LABEL   far
```

Please see also: Forward References (9.2.4)

9.3.8 SEG Operator

Syntax:

```
SEG expression
```

The SEG operator returns the segment selector value for *expression*. The resulting value is relocatable and is 16 bits wide, and its final value is not determined until the program is loaded into memory for execution.

For 8086, 8088, 80186, or 80188 programs (or for 80286 or 80386 real mode programs), the segment selector is the segment paragraph address — i.e., the address of the 16 byte boundary where the segment is located. To turn a segment paragraph address into a byte address, it must be shifted left four bits to create a 20 bit wide byte address.

For 80286 or 80386 protected mode programs, the segment selector is an index of a segment descriptor in a table maintained by the operating system. Each entry in the descriptor table identifies the address where the segment is located, its length, and other information about the segment.

Example:

```
mov    ax,SEG var1    ; load selector for segment variable
                      ; var1 is located in
mov    ax,SEG lab1    ; load selector for segment label
                      ; lab1 is located in
```

Please see also: Chapter 11

9.3.9 Structure Field Operator

Syntax:

expression.strucfldname
expr1.expr2

The structure field operator (.) is used to index to a field within a structure. It adds the offset of the structure field within the structure to *expression,* and converts the data type of *expression* to the data type of the structure field.

The structure field operator may also be used as a synonym for + to add two expressions.

Example:

```
sdef1   STRUC
        sfld1     DB      0
        sfld2     DW      0
sdef1   ENDS

strucl  sdef1     <1,2>               ; allocate storage for a
                                      ; structure

        mov       al,strucl.sfld1     ; load byte at offset 0
                                      ; in strucl
        mov       bp,OFFSET strucl    ; load pointer to
                                      ; strucl
        mov       ax,[bp].sfld2       ; load word at offset 1
                                      ; in strucl
```

Please see also: Referencing Structures (6.8)

9.3.10 Segment Override Operator

Syntax:

```
segregister:expression
segname:expression
groupname:expression
```

The segment override operator (:) is used to aid the assembler in producing correct object code for memory references which do not use the default segment register. By default, references to data use DS, SS, or ES as the default segment register, depending on the instruction (the vast majority of references assume DS, stack instructions assume SS, and some string move instructions assume ES).

If an instruction references data in a segment that is not pointed to by the default segment register assumed for that instruction, a segment override byte must precede the instruction. This segment override byte tells the processor which segment register to use to access the data. 386 | ASM automatically generates a segment override byte for references to symbols that are defined before the line that references them. If a symbol is forward referenced, and it is in a segment not pointed to by the default segment register, the segment override operator must be used to tell 386 | ASM that a segment override byte is required.

If a segment name or group name is used with a segment override, the assembler must have previously been informed (with the ASSUME directive) which segment register will point to the segment or group, so that it can generate the correct override byte.

Note that it is possible to use a segment override to a segment other than the segment in which a variable is defined. This may be necessary, for example, if no segment register is set up to point to the segment the variable is defined in. In this case, the assembler not only outputs a segment override byte if necessary; it also forces the relocatable offset of the variable to be computed relative to the segment used in the segment override, rather than to the segment in which it is defined. This is necessary in order for the processor to reference the correct location; since the segment register used points to the segment used in the segment override, the offset must also be relative to that segment. This type of

segment override cannot be used with 80286 or 80386 protected mode programs.

Example:

```
        ASSUME  cs:cseg,ds:dseg1,es:dseg2

dseg1   SEGMENT
var1    DW      0
dseg1   ENDS

dseg2   SEGMENT
var2    DW      0
dseg2   ENDS

cseg    SEGMENT

var3    DW      0

        mov     ax,var1          ; no segment override
                                 ; necessary
        mov     ax,var2          ; segment override to ES
                                 ; automatically generated by
                                 ; 386|ASM
        mov     ax,var3          ; segment override to CS
                                 ; automatically generated by
                                 ; 386|ASM
        mov     ax,forw1         ; no segment override
                                 ; necessary
        mov     ax,es:forw2      ; 386|ASM must be told
                                 ; that a segment
                                 ; override is necessary
        mov     ax,dseg2:forw2   ; another way of doing
                                 ; the same thing
        mov     ax,cs:forw3      ; 386|ASM must be told
                                 ; that a segment
                                 ; override is necessary
        mov     bp,OFFSET var2   ; load pointer to var2,
                                 ; relative to ES
        mov     ax,es:[bp]       ; 386|ASM must be told
                                 ; that a segment
                                 ; override is necessary

forw3   DW      0

cseg    ENDS
```

```
dseg1   SEGMENT
forw1   DW      0
dseg1   ENDS

dseg2   SEGMENT
forw2   DW      0
dseg1   ENDS
```

Please see also: Chapter 11, Forward References (9.2.4), OFFSET (9.3.11), ASSUME (3.10)

9.3.11 OFFSET Operator

Syntax:

`OFFSET` *`expression`*

The OFFSET operator returns the offset of *expression* relative to the segment in which it is defined. It is useful for loading the effective address of a label or variable into a register.

Note that if a segment override is used in the expression, the offset returned is relative to the segment specified in the override, rather than to the segment in which the variable or label is defined. This is necessary to compensate for a shortcoming in the assembly language definition. If a variable is defined in a segment that is within a group, and the group is pointed to by a segment register, memory references to the variable correctly relocate the variable offset relative to the group. However, the OFFSET operator returns the offset of the variable within the segment rather than within the group, so a segment override to the group must be used to obtain the correct offset value.

If the OFFSET operator is used on a non-relocatable value, it returns the value of the expression. This feature can be used for code clarity when a structure field offset must be used as an immediate value.

Example:

```
        ASSUME  cs:cseg,ds:dgroup

dgroup  GROUP dseg1,dseg2

dseg1   SEGMENT
```

```
var1     DW     0
dseg1    ENDS
str1     STRUC
field1   DB     ?
field2   DW     ?
field3   DD     ?
str1     ENDS

dseg2    SEGMENT
var2     DW     0
dseg2    ENDS

cseg     SEGMENT

         mov    ax,var2    ; offset correctly relocated
         lea    bp,var2    ; offset correctly relocated
         mov    ax,[bp]

         mov    bp,OFFSET dgroup:var2   ; segment override
         mov    ax,[bp]                 ; must be used to
                                        ; make offset be
                                        ; correctly
                                        ; relocated

         mov    ax,OFFSET lab1   ; offset relocated
                                 ; correctly
         jmp    ax
lab1:

         mov    ax,OFFSET field2        ; offset of
                                        ; structure field

cseg     ENDS
```

Please see also: SEGMENT (3.7), GROUP (3.9), Segment Override (9.3.10)

9.3.12 SHORT Operator

Syntax:

SHORT *label*

The SHORT operator instructs 386 | ASM to generate a short (eight byte) offset to a label in a jump instruction. The label must be no more than

127 bytes from the jump instruction, or an error will be generated. Judicious use of the SHORT operator allows the assembler to generate less code. It is only necessary to use the SHORT operator if the label is being forward referenced. For references to labels defined above, 386|ASM automatically generates a short offset, if possible.

Example:

```
lab1:
        jmp     lab1            ; short offset
                                ; automatically generated
        jmp     SHORT lab2      ; tell assembler to
                                ; generate short offset
lab2:
```

9.3.13 THIS Operator

Syntax:

`THIS` *`datatype`*

The THIS operator creates an operand with the specified data type, with a relocatable value equal to the current offset within the current segment. The *datatype* parameter must be one of the assembler reserved words given in Table 9–3.

Example:

```
        mov    al,THIS byte    ; load first byte of MOV
                               ; opcode
lab1    equ    THIS near       ; equivalent to lab1: or
                               ; lab1 LABEL near
```

9.3.14 .TYPE Operator

Syntax:

`.TYPE` *`expression`*

The .TYPE operator returns a constant value giving information about the expression. Table 9–7 shows the defined bits in the returned value. All other bits in the returned value are guaranteed to be zero.

TABLE 9-7
BIT DEFINITIONS FOR .TYPE OPERATOR

Bit	Mask	Meaning
0	01h	If the bit is 1, the expression has a label data type. Otherwise, the bit is 0.
1	02h	If the bit is 1, the expression has a variable data type. Otherwise, the bit is 0.
5	20h	If the bit is 1, there are no undefined symbols in the expression. If there is at least one undefined symbol in the expression, then this bit and all other bits in the returned value are set to 0.
7	80h	If the bit is 1, there is a symbol that is defined external in the expression. Otherwise, the bit is 0.

9.3.15 Indirection Operator

Syntax:

```
[effective address expression] expr1[expr2]
```

The indirection operator, `[]`, is used to indicate indirection through a register in effective address expressions. This is described in detail in Section 9.4. It can also be used as a synonym for the + operator in order to add two expressions.

Note that operations within brackets are always calculated before adjacent operations, just as with expressions in parentheses.

Example:

```
array   DB    20 DUP(?)

        mov   ax,[bp]          ; indirection through BP
                               ; register
        mov   ax,10[array]     ; load the 11th element in
                               ; array
```

9.4 Effective Address Modes

Addressing modes on the 8086 family of processors use the concepts of base and index registers, displacements, and scale factors on index registers. A base or index register is used in register indirect addressing modes. A displacement is an 8, 16, or 32 bit immediate value which is added to an effective address calculated with one of the register indirect addressing modes. A displacement may be either a constant or relocatable value. A scale factor is a value of one, two, four, or eight that is multiplied by the contents of an index register before it is used to calculate the effective address. Table 9–8 identifies which registers may be used as base or index registers. Please see Appendix F for a summary of all the programmer–accessible registers on the 80386.

TABLE 9-8
BASE AND INDEX REGISTERS IN THE 8086 FAMILY

Register Type	8086,8088, 80186,80286	80386
Base Register	BX,BP	BX,BP, any 32 bit general register
Index Register	SI,DI	SI,DI, any 32 bit general register except ESP

9.4.1 Immediate Operand Mode

Syntax:

constant_expression

Immediate operand mode specifies that the instruction operand is encoded as part of the instruction.

Example:

```
var1  DW     0
      mov    ax,1              ; the value one is an
                               ; immediate operand
      mov    ax,OFFSET var1    ; the offset value is
                               ; an immediate
                               ; operand
```

9.4.2 Memory Direct Mode

Syntax:

relocatable_expression

In memory direct mode, the address of the instruction operand is encoded as part of the instruction.

Example:

```
var1  DW   0
      mov  ax,var1    ; the operand is the data
                      ; located at var1 the address of
                      ; var1 is given in memory direct
                      ; mode
```

9.4.3 Register Direct Mode

Syntax:

register

In register direct mode, the operand of the instruction is the contents of a register. Any general register, plus the segment registers and the 80386 control, test, and debug registers, may be used in register direct mode.

Example:

```
mov    ax,bx    ; The operand is the contents of BX
```

9.4.4 Register Indirect Mode

Syntax:

```
[register]
displacement[register]
displacement + [register]
[register + displacement]
[register].displacement
[register] + displacement
```

In register indirect mode, the register contents are added to an optional immediate displacement to obtain the address of the instruction operand. Any register which may be used as a base or index register may be used in this address mode.

Example:

```
array   DB      20 DUP (0)
        mov     ebx,OFFSET array
        mov     al,[ebx]          ; register indirect
        mov     al,10[ebx]        ; register indirect with
                                  ; displacement
        mov     al,[ebx+10]
        mov     al,[ebx].10

        mov     ebx,10
        mov     al,array[ebx]     ; register indirect with
                                  ; relocatable
                                  ; displacement
```

9.4.5 Based Index Mode

Syntax:

```
[basereg][indexreg]
displacement[basereg][indexreg]
displacement + [basereg][indexreg]
[basereg + displacement][indexreg]
[basereg][indexreg + displacement]
[basereg][indexreg].displacement
[basereg][indexreg] + displacement
```

In based index address mode, the effective address of the instruction operand is given by the sum of the contents of the base and index registers and the optional immediate displacement.

Example:

```
array   DB      20 DUP (0)
        mov     bp,OFFSET array
        mov     si,10
        mov     al,[bp][si]       ; based index addressing
        mov     al,[bp][si].2     ; based index with
                                  ; displacement
```

9.4.6 Scaled Index Mode

Syntax:

```
[indexreg * constant_expression]
displacement[indexreg * constant_expression]
displacement + [indexreg * constant_expression]
[indexreg * constant_expression].displacement
[indexreg * constant_expression] + displacement
```

In scaled index mode, the address of the instruction operand is given by the contents of the index register multiplied by a scale factor of one, two, four, or eight, and then added to an optional immediate displacement. This address mode is only available on the 80386.

Example:

```
array   DD      20 DUP (0)
        mov     ecx,2                 ; 3rd entry in array
        mov     eax,array[ecx*4]      ; scaled index mode
```

9.4.7 Based Scaled Index Mode

Syntax:

```
[basereg][indexreg * constant_expression]
displacement[basereg][indexreg * constant_expression]
displacement+[basereg][indexreg * constant_expression]
[basereg+displacement][indexreg * constant_expression]
[basereg][indexreg * constant_expression].displacement
[basereg][indexreg * constant_expression]+displacement
```

In based scaled index mode, the address of the instruction operand is given by the contents of the index register multiplied by the scaled factor, and then added to the contents of the base register and an optional immediate displacement. This address mode is only available on the 80386.

Example:

```
array   DD      20 DUP (0)
        mov     ebx,offset array
        mov     ecx,2                 ; 3rd entry in array
        mov     eax,[ebx][ecx*4]      ; based scaled index
                                      ; mode
        mov     ax,2[ebx][ecx*4]      ; load high byte of
                                      ; array entry
```

Macros and Repeat Blocks

10.1 Introduction

A macro is a named block of source code statements that is created with the MACRO directive. Once a macro is defined, it can be expanded by referencing it by name. When a macro is expanded, copies of the source code statements in the macro definition are inserted in the source code being assembled. Macros may also have named parameters, for which substitution values are specified when the macro is expanded.

A repeat block is a sequence of source code statements that is replicated a specific number of times under the control of the repeat block directive being used. Unlike macros, there is no concept of a separate definition and expansion; the block is repeated in place.

10.2 Macro Definition

Syntax:

```
name MACRO {formalparam,...}
statements
ENDM
```

A macro definition is created by surrounding the block of statements in the macro definition by the MACRO and ENDM directives. The macro is given a required *name* and an optional list of formal parameter names. When the macro is expanded, each occurrence of the formal parameter names in the statement block is replaced by the actual parameter values specified when the macro is invoked. The formal parameter names are kept with the macro definition and therefore will never conflict with other symbols defined elsewhere in the file. The macro name is a normal user-defined symbol and must be unique. There are no limits imposed on the number of formal parameters or the number of source code

statements in a macro definition, other than the requirement that all the formal parameter names must be listed on one line.

No object code is generated when a macro is defined, but the macro definition is saved in memory. Object code is generated when the macro is expanded further on in the source file. A macro definition can therefore be placed outside a segment block, if desired.

The following example creates a macro definition named alloc with two formal parameters, val1 and val2, and with two source code statements.

```
alloc  MACRO   val1,val2
       DB      val1
       DB      val2
       ENDM
```

It is legal to redefine a macro. When the assembler encounters a new definition for a macro, it discards the old definition and saves the new one. Macro expansions after the first macro definition, but before the second, will use the first macro definition. Macro expansions after the second macro definition use the second definition.

10.3 Macro Expansions

Syntax:

```
name {actualparam, ...}
```

A macro is expanded when a previously defined macro is invoked by *name*. All of the source statements in the macro definition are inserted at that point in the source file and assembled to generate object code. Each occurrence of a formal parameter name in a source statement is replaced by the corresponding actual parameter from the macro invocation statement. Actual parameters are treated as character strings, and are separated by commas, spaces, or tabs. If there are fewer actual parameters than formal parameters, the missing actual parameters are assigned the null string. A null string may be passed for any particular actual parameter by entering two commas in a row.

When a macro is expanded, there are three options available for what will be printed in the listing file: all of the inserted statements will be printed, only statements that cause object code to be generated will be printed, or none of the inserted statements will be printed. The desired option can be selected by using the appropriate listing file directives (please see sections 8.3.11–8.3.13). By default, only statements for which object code is generated are printed.

The following examples show two statements invoking the alloc macro that was defined in section 10.2 and the source code statements that will be inserted.

```
alloc   0,1
```

will expand to:

```
DB      0
DB      1
```

the statement:

```
alloc   "that's" 'show biz'
```

will expand to:

```
DB      "that's"
DB      'show biz'
```

Macros may also be recursive, and macro expansions may be nested. For example, the following pair of macro definitions:

```
term    MACRO
        DB      0 ;; Terminate the string with a zero
                  ;; byte
        ENDM

bldstg  MACRO   val
        IFIDN   <val>,<5>
        term                    ;; terminate the string
        ELSE
        DB      'val&'          ;; next character in
                                ;; string
        bldstg  %val - 1        ;; continue loop
        ENDIF
        ENDM
```

will expand to:

```
bldstg  9
DB      '9'
DB      '8'
DB      '7'
DB      '6'
DB      0
```

Please see also: .LALL (8.3.11), .SALL (8.3.12), .XALL (8.3.13), Conditional Assembly (8.4), Parameter Substitution (10.11)

10.4 REPT Repeat Block

Syntax:

```
REPT expression
statements
ENDM
```

The REPT directive causes the block of statements surrounded by the REPT and ENDM directives to be repeated *expression* number of times, where *expression* must evaluate to a constant value. For example, the repeat block:

```
REPT    3
DB      0
ENDM
```

will expand to:

```
DB      0
DB      0
DB      0
```

10.5 IRPC Repeat Block

Syntax:

```
IRPC formalparam,string
statements
ENDM
```

The IRPC directive causes the block of statements enclosed by the IRPC and ENDM directives to be repeated as many times as there are characters in *string*. Each time the statement block is repeated, the next character in *string* is substituted for *formalparam*. For example, the repeat block

```
IRPC    val,012
DB      val
ENDM
```

will expand to:

```
DB      0
DB      1
DB      2
```

Please see also: Conditional Assembly (8.4)

10.6 IRP Repeat Block

Syntax:

```
IRP formalparam,<actualparam, ...>
statements
ENDM
```

The IRP directive causes the block of statements enclosed by the IRP and ENDM directives to be repeated once for each actual parameter in the actual parameter list. Each time the statement block is repeated, the current *actualparam* is substituted for *formalparam*. The syntax for the actual parameter list is exactly the same as it is for the actual parameter list in a macro invocation. The actual parameter list must be enclosed in angle brackets. For example, the repeat block:

```
IRP     val,<0, 'string', 1>
DB      val
ENDM
```

will expand to:

```
DB      0
DB      'string'
DB      1
```

Please see also: Parameter Substitution (10.11), Conditional Assembly (8.4)

10.7 Macro and Repeat Block Comments

Syntax:

```
;; comment
```

A comment which begins with two semicolons inside a macro or a repeat block definition is called a macro/repeat block comment. Macro/repeat block comments do not appear in the expansion of the macro or repeat block. They are used to document the macro definition or repeat block without cluttering up the expansion listing with duplicate comments. For example, the following macro definition:

```
define  MACRO   val1,val2
        DB      val1  ; this comment will show up
        DB      val2  ;; this one won't
        ENDM
```

will expand to:

```
define  0,1
DB      0       ; this comment will show up
DB      1
```

10.8 LOCAL Directive

Syntax:

```
LOCAL formalparam, ...
```

The LOCAL directive substitutes an automatically generated name of the form ??XXXX, where XXXX is a hexadecimal number, for *formalparam* each time the macro is expanded. If more than one formal parameter is given, they must be separated by commas. The LOCAL directive can only

be used within a macro definition block; it cannot be used within a repeat block.

For example, given the macro definition:

```
chk_err MACRO   limit
        LOCAL   skip
        cmp     ax,limit         ;; check value against
                                 ;; limit
        jle     skip             ;; skip call if OK
        call    error            ;; call error procedure
skip:
        ENDM
```

The following macro invocations:

```
chk_err 5
chk_err 10
```

will cause the following code to be expanded:

```
        cmp     ax,5
        jle     ??0000
        call    error
??0000:
        cmp     ax,10
        jle     ??0001
        call    error
??0001:
```

10.9 EXITM Directive

Syntax:

```
EXITM
```

The EXITM directive terminates expansion of a macro or a repeat block immediately. For example, the following repeat block:

```
IRPC    val,0123456789
IFIDN   <val>,<3>
EXITM
ELSE
DB      val
ENDIF
ENDM
```

will cause the following statements to be generated:

```
DB      0
DB      1
DB      2
```

Please see also: Conditional Assembly (8.4)

10.10 PURGE Directive

Syntax:

PURGE *name,...*

The PURGE directive has no effect and is present only so that no errors will be generated by PURGE statements in existing 8086 programs.

10.11 Parameter Substitution

Syntax:

&formalparam
formalparam&
&formalparam&

When a macro is expanded, 386 | ASM substitutes actual parameters for each of the formal parameters in the macro definition. Each occurrence of a formal parameter name, which is separated from adjacent symbols by spaces, tabs, or the ampersand character (&), is replaced with its corresponding actual parameter. The ampersand may be placed on either side or on both sides of the formal parameter name and is removed along with the formal parameter name when the actual parameter is

substituted. It is also used to force parameter substitution inside a string constant. For example, the following repeat block:

```
         IRP        val,<0 1 2>
var&val  DB         'This value is &val&'
         ENDM
```

will expand to:

```
var0   DB     'This value is 0'
var1   DB     'This value is 1'
var2   DB     'This value is 2'
```

If two formal parameters are placed adjacent to each other, they must be separated by two ampersand characters, in order to force substitution of both parameters. This happens because each substitution removes one ampersand character. This is true even if the formal parameters are parameters for two separate macros in nested macro expansions or repeat blocks. For example, the following macro definition with a nested repeat block:

```
define   MACRO    a,b
         DB       'a&&b'
         IRPC     c,012
a&&c     DB       b
         ENDM
         ENDM
```

will expand to:

```
         define   test 32
         DB       'test32'
test0    DB       32
test1    DB       32
test2    DB       32
```

Please see also: Macro Expansions (10.3), IRPC (10.5), IRP (10.6), Conditional Assembly (8.4)

10.12 Actual Parameter Lists

Lists of actual parameters passed to macro expansions and lists of actual parameters for the IRP directive follow the same syntax. An individual

actual parameter is a string of characters that is terminated by a space, a tab, a comma, a semicolon, or a left angle bracket (<). In addition, there are special operators that are valid only in actual parameter lists to permit any character to become part of an actual parameter.

Please see also: Macro Expansions (10.3), IRP (10.6)

10.12.1 Literal Character Operator

Syntax:

```
!character
```

The literal character operator (!) causes the next character to be treated as a literal character when the actual parameter list is processed. It can be used to escape characters that would normally have a special meaning (parameter separators or parameter list operators) to make them a part of the parameter.

Example:

```
define  MACRO   string
        DB      '&string'
        ENDM

        define  parameter! with! spaces
        DB      'parameter with spaces'

        define  a!;b; parameter with a ; in it
        DB      'a;b'

        define  0! !<! 1        ; escape < operator
        DB      '0 < 1'
        define  My! Gosh!!         ; escape ! operator
        DB      'My Gosh!'
```

10.12.2 Literal Text Operator

Syntax:

```
<text>
```

The literal text operator (<>) causes all the text within the angle brackets to be treated as literal text when the actual parameter list is processed. It creates a single actual parameter whose value is *text*. Any previous actual parameter on the line is terminated by the < character, and this actual parameter is terminated by the > character. Like the literal character operator, this operator can be used to escape separator characters in parameter lists. This operator should not be confused with the angle brackets which are required to surround the actual parameter list for the IRP directive (please see section 10.6).

The only characters within angle brackets which are not treated as literal text are the other two parameter list operators (! and %) and any angle brackets. The ! and % operators are still active, and any angle brackets within the text must be matched (e.g., <abc<de>fg> yields a string "abc<de>fg"). If unmatched angle brackets are desired within the text, the literal character operator must be used (e.g., <abc!<defg> yields a string "abc<defg").

Example:

```
define  MACRO   list
        DB      list
        ENDM

        define  <1,2,3>         ; Escape comma separator
        DB      1,2,3

define  MACRO   str1,str2
        DB      '&str1&'
        DB      'str2&'
        ENDM

        define  this< is a string with spaces>
        DB      'this'
        DB      ' is a string with spaces'

        define  this,< is a <string> with angle brackets>
        DB      'this'
        DB      ' is a <string> with angle brackets'

        define  this < is a string with a !>>
        DB      'this'
        DB      ' is a string with a >'
```

10.12.3 Expression Operator

Syntax:

```
%expression
```

The expression operator evaluates an *expression* to obtain a constant value, converts the value into an ASCII decimal string, and gives the actual parameter the resulting string as a value. The string that is evaluated as an expression is terminated by a comma, a semicolon, an exclamation point, a left or right angle bracket, or another percent character (i.e., one of the set ,;!<>%).

Normally the expression operator can only be used to return a single actual parameter. It must be the first character in the actual parameter. If a % is encountered in a string of text, it is normally treated as any other text character. The exception to this rule is that the % operator is recognized anywhere within literal text enclosed in angle brackets (<>).

Example:

```
define  MACRO   val,string
        DB      val
        DB      '&string'
        ENDM

        define  %1 + 2,=3
        DB      3
        DB      '=3'

        define  % 1 + 2 %3+4
        DB      3
        DB      '7'

        define  %3*4-2 <=10>
        DB      10
        DB      '=10'

        define  %3*4-2 !!=! 5
        DB      10
        DB      '!= 5'

        define  %1 + 2,<is equal to %1 + 2, right?>
        DB      3
```

```
        DB      'is equal to 3, right?'

        define  1,99%of100      ; notice % within string
                                ; has no effect
        DB      1
        DB      '99%of100'
```

10.12.4 String Operators

Syntax:

```
'string'
"string"
```

Single or double quotes may be used as string operators in an actual parameter to cause all characters within the quotes, including special characters such as spaces, commas, and other parameter list operators, to be treated as literal text. The string operators differ from the literal text operator in three ways: (1) The quotes that delimit the string are kept as part of the actual parameter. (2) All characters are treated literally within the string, including other parameter list operators. (3) The string operators may be embedded in the text of an actual parameter; they do not automatically terminate the previous actual parameter and begin a new one.

Example:

```
define  MACRO   str1,str2
        DB      str1
        DB      'str2&'
        ENDM

        define  "lot's! of <special>,chars%", 0
        DB      "lot's! of <special>,chars%"
        DB      '0'

        define  0, one" way to get "spaces
        DB      0
        DB      'one" way to get "spaces'
```

Chapter 11

Programming the 80386

This chapter describes some of the unique aspects of programming members of the 8086 processor family. A basic knowledge of assembly language programming and of the 80386 processor architecture is assumed. The purpose of this chapter is to aid someone who already knows how to write assembly language programs for other machines to quickly begin writing programs for the 80386. It is **not** a tutorial on how to program in assembly language.

11.1 Real Mode Programming

The 80386 has two primary modes of operation: real mode and protected mode. In real mode, the processor essentially looks like a fast 8086 processor. Current versions of MS-DOS execute only in real mode, so the 80386 processor is normally executing in real mode if the operating system is MS-DOS. This section discusses programs written to execute in real mode on the 80386 or 80286 (which also has real and protected modes of operation), and programs written to execute on an 8086, 8088, 80186, or 80188 processor. To assemble and link a program in a source code file named HELLO.ASM for execution in real mode, type:

```
386asm hello -8086
386link hello -8086
```

This assembles the source file HELLO.ASM to create an object file HELLO.OBJ and a listing file HELLO.LST, and then links the object file to create an executable task image HELLO.EXE and a map file HELLO.MAP.

An important feature of the 8086 family architecture is the concept of segmentation. In real mode, a program is partitioned into segments, each of which may be up to 64 kilobytes (KB) in length. The processor has a special set of registers, called segment registers (please see Appendix F),

which must be set up to point to a segment before data or code in the segment can be accessed or executed.

In real mode, a segment register contains the paragraph address of a segment, where a paragraph is a 16 byte boundary within the address space. A full address within a segment is given by the segment paragraph address and by an offset (from 0 to 64 KB) within the segment. The processor creates an internal 20 bit address from this by shifting the segment address left four bits, turning it into a byte address, and then adding it to the offset within the segment. Note that this method of creating an internal address limits the address space of the processor in real mode to one megabyte (an internal address is 20 bits wide). For reasons discussed in the next section, we call the value in the segment register a segment selector; and we call the internal address created by the processor from the segment selector and the segment offset a linear address. On the 8086, 8088, 80186, 80188, or on the 80286 or 80386 executing in real mode, the linear address is always the same as the physical address that the processor puts out on the address bus to access the memory.

Specific segment registers are assumed for specific operations performed by the processor. The CS register is assumed (by the processor) to point to the current code segment; the DS register is assumed to point to the current data segment, the SS register to the current stack segment, and ES, FS, and GS point to extra data segments. Most programs must have CS, DS, and SS set up to point to valid segments in order to execute. It is not necessary to set up ES, FS, or GS unless the program makes specific use of them.

For every memory reference made by the processor, there is a default segment register to compute the internal memory address. For instruction fetch operations, CS is assumed. For data references, DS is assumed. For stack operations (e.g., PUSH, POP, and indirect references using the EBP or ESP register) SS is assumed. ES is only assumed for the destination of some string operations, and FS and GS are never assumed. Please see Appendix D and the list of related documents in the the list of related documents in the Preface for specific details on which operations assume which segment registers.

It is possible to override the segment register that the processor uses for a memory reference on an individual instruction by prefixing the

instruction opcode with a segment override byte. This is done automatically by 386|ASM for memory references which do not obey the defaults used by the processor. For example, a data reference to a variable in the segment pointed to by CS would have a segment override byte to tell the processor to use the CS register instead of the DS register in computing the variable's address. Obviously, in order to correctly generate override bytes, 386|ASM must know which segments are pointed to by which segment registers. This is done with the ASSUME directive (please see section 3.10) and is a necessary part of an assembly language source code file.

When a program is loaded for execution by MS-DOS, the CS segment register is initialized by the program loader to point to the code segment where execution begins. The DS register is initialized by the loader to point to a block of information about the program (**not** to the program's data segment). Initalizing DS to point to its data segment is usually the first operation performed by the program when it begins execution. In addition, all segments with a combine type of STACK (please see section 3.7.2) are combined by the linker into a single segment. The loader initializes SS to point to this segment, and SP (the stack pointer) to point to the highest offset within the segment (the stack grows downward in the 8086 family). If there is no segment in the program with a STACK combine type, SS and SP will not be initialized by the loader; it is then the responsibility of the program to initialize those registers as well as DS.

Typically, an application program will have three segments: a code segment, a data segment, and a stack segment. Frequently, the stack and data are combined into a single segment which is pointed to by both DS and SS, with the data at the bottom of the segment and the stack at the top of the segment. In simple programs, it is even possible to have only one segment, with CS, DS, and SS all pointing to the same segment.

Large programs may require more than one data segment, or more than one code segment, or both. (Remember that a single segment in real mode is limited to 64 KB in length.) Multiple data segments are handled either by setting up DS to point to whichever data segment you need to access, or by keeping DS pointing at the main data segment and using the extra data segment registers ES, FS, and GS to access the other data segments. Remember that every time a segment register is reloaded with

a new value, 386|ASM must also be informed of the change with the ASSUME directive so that it can generate correct code!

Multiple code segments are handled by having two forms of call and jump instructions: a NEAR call or jump, and a FAR call or jump. NEAR control transfers are within the segment currently pointed to by the CS register and have an offset within the segment as an operand to the instruction. FAR control transfers are to another segment and have as operands both a segment selector (in real mode, the segment paragraph address) and an offset within the new segment. When the FAR call or jump is executed, the processor loads CS with the new segment selector given by the instruction. 386|ASM automatically generates the correct form for a call or jump instruction. It determines which form to use from the data type (NEAR or FAR) of the label or procedure which is the destination of the call or jump (please see sections 5.2 and 5.3.1). Note that there are also two forms of the return instruction. 386|ASM generates the correct form by generating a FAR return within a FAR procedure, and a NEAR return within a NEAR procedure. Once again, an ASSUME directive must be placed before each code segment in an assembly language source file to tell 386|ASM that the CS register will point at that segment during program execution.

Note that the only legal way to change the contents of the CS register is with a FAR call or jump, or with one of the return instructions which pops the CS register. The two return instructions which pop the CS register are the FAR form of the RET instruction, and the return from interrupt (IRET) instruction.

The following simple example program demonstrates some of these concepts. It performs an MS-DOS system call to output the message "Hello world....." on the terminal when it is executed. This program has three segments: a code segment which is pointed to by the CS register, a data segment which is pointed to by the DS register, and a stack segment which is pointed to by the SS register. Since the program does not use the ES, FS, and GS registers, it does not even bother to initialize them. At first glance, it may appear that this program is so simple that it does not even use the stack segment. In fact, the INT (software interrupt) instruction pushes a flags word and a return address on the stack. Hardware interrupts during normal CPU operation also require space on the stack

for a return address. Thus, every program must set up the SS and SP registers to point to a valid stack area in memory.

```
TITLE   hello.asm

 ;
 ; The .8086 directive can be used instead of the
 ; -8086 command line switch to cause programs to
 ; be assembled for execution in real mode on the
 ; 80386 or 80286, or for execution on the 8086,
 ; 8088, or 80186.  The -8086 command line switch
 ; must still be used with the linker, however!
 ;
         .8086

 ;
 ; Tell the assembler which segments the CS, DS,
 ; and SS registers will point to
 ;
         ASSUME  cs:cseg,ds:dseg,ss:sseg



 ;
 ; Align the data segment on a paragraph (16 byte)
 ; boundary and give it the standard combine type
 ; of PUBLIC.
 ;
 data    SEGMENT PARA PUBLIC 'DATA'

 ;
 ; Message to be output, formatted as required by
 ; the MS-DOS system function call
 ;
 hellomsg DB     'Hello world.....',0Dh,0Ah,'$'

 data    ENDS

 ;
 ; Allocate 1000 bytes in a segment with a
 ; combine type of STACK, which will cause the
 ; loader to initialize SS to point to it,
 ; and initialize SP to the highest offset.
 ;
 sseg    SEGMENT PARA STACK 'STACK'
```

```
        DB      1000 DUP (?)

sseg    ENDS

;
; Also give the code segment the standard
; combine type of PUBLIC.
;
cseg    SEGMENT PARA PUBLIC 'CODE'

;
; The actual code to be executed is in
; a NEAR procedure.
;
print   PROC    NEAR

;
; Initialize DS to point to the data segment
;
        mov     ax,dseg
        mov     ds,ax

;
; Load the offset within the segment pointed
; to by DS of the message to be output, load
; the MS-DOS function code into AH, and trap
; to the operating system to output the
; message to the terminal.
;
        mov     ah,09h
        mov     dx,OFFSET hellomsg
        int     21h

;
; Load the MS-DOS terminate process function
; code, and trap to the operating system
;
        mov     ah,4ch
        int     21h

print   ENDP

cseg    ENDS

;
; Identify the entry point of the program
; where the loader will transfer control after
```

```
; it loads the program and initializes the
; CS, SS, and SP registers.
;
        END     print
```

This is a complete program. On 80386, 8086, 8088, or 80286 systems running MS-DOS, you can create a file with this code in it, assemble and link it, as shown in the example at the beginning of this section, and then execute it to see it write the message "Hello world....." on your terminal.

Other than segmentation, the only feature of the 8086 processor family that is not in common use on most other processors is the use of dedicated registers by some instructions. For most instructions, the general registers (please see Appendix F) can be used interchangeably. However, some instructions assume that specific registers contain operands for the instruction. For example, the CX register contains a count operand for some string instructions. The AX, BX, and DX registers also have specialized uses for certain instructions. Instructions which dedicate certain registers for specific uses are identified in Appendix D and in the list of related documents in the Preface.

11.2 Protected Mode Programming

Protected mode is the mode of operation that permits use of the full power of the 80386. It allows access to a much larger address space, permits individual segments to be much larger (thereby eliminating the difficulties of dealing with multiple segments for large programs), and supports 32 bit wide operands for arithmetic operations and use of the 32 bit wide register set of the 80386. To assemble and link a simple program in a file called HELLO.ASM for protected mode operation on the 80386, type the following commands:

```
386asm hello
386link hello
```

The main difference between real mode and protected mode on the 80386, from the point of view of the application program, is the way in which segmentation is handled. In protected mode, an offset within a segment is 32 bits wide, so a single segment can be up to four gigabytes long. Segments are controlled by two kinds of tables that are maintained by the

operating system: a global descriptor table (GDT), of which there can only be one per system; and a local descriptor table (LDT), of which there can be one per process (and therefore many in a multiprocessing system). A segment register contains a value called a segment selector, which contains a bit which selects either the GDT or the LDT for the current process. Two bits are used to control access to privileged segments, and 13 bits are used to index to an entry in the selected descriptor table. A GDT or LDT can therefore contain up to 8192 entries. Each entry is eight bytes in size, and gives the starting address for the segment, and the size of the segment. It also gives status information which identifies the segment as either a code or a data segment, and identifies the privilege level of the segment, etc. Please see the list of related documents in the Preface for a complete description of how segmentation is handled in protected mode.

A linear address in protected mode is created by the processor from the segment selector and the offset within the segment as follows: The 16 bit wide segment selector value (in the segment register) is used to identify an entry in one of the descriptor tables. This entry gives a 32 bit wide linear base address for the segment. The segment base address is then added to the 32 bit wide segment offset to yield a 32 bit wide linear address. The processor does some internal caching of descriptor table entries to speed up this process, but this is invisible to the programmer.

The 80386 processor also supports paging on the chip. This is the reason the address formed from the segment selector and segment offset is called a linear address rather than a physical address. Paging can be disabled in protected mode, in which case the linear address is the same as the physical address the processor puts out on the address bus. If paging is enabled, the paging unit on the chip will translate the 32 bit wide linear address into a 32 bit wide physical address. This is invisible to the programmer and can be ignored; for a complete description of paging, please see the list of related documents in the Preface.

Protected mode segments must be designated as either code segments or data segments. Code segments may not be written to, and code in data segments may not be executed. It therefore appears that any protected mode program must have at least two segments: a code segment and a data segment. In fact, it is possible to have a program with a single segment by creating two entries in the descriptor table for a single segment. One entry identifies the segment as a code segment, and the

other identifies the segment as a data segment. The CS register is loaded with the selector for the code segment entry in the descriptor table, and the DS register is loaded with the selector for the data segment entry.

The important differences between protected mode and real mode, from the point of view of the application programmer, can be summarized as follows:

1. The value in a segment register is no longer an address, as it is in real mode. Instead, it is an index to an entry describing the segment in a descriptor table.

2. An offset within a segment is 32 bits wide, instead of 16 bits wide. An individual segment can therefore be up to four gigabytes in length, as opposed to 64 kilobytes in real mode.

3. A linear (and a physical) address is 32 bits wide, versus 20 bits wide in real mode. Up to four gigabytes of memory therefore can be directly addressed in protected mode, as opposed to one megabyte in real mode.

4. Operands to instructions can be 32 bits wide. (This is important for arithmetic operations in particular). A 32 bit wide set of general registers is available (please see Appendix F).

The following simple program is written to execute in protected mode on the 80386. It is the HELLO.ASM program from the example above, adapted a bit for protected mode. The basic differences are that the segments are set up as USE32 segments (this is the default for the 80386, and its meaning is described below the example program), and a couple of contrived instructions in the program show off the use of 32 bit registers and an instruction added for the 80386.

```
        TITLE   hello.asm

;
; The .386c directive is actually unnecessary,
; since it is the default for 386|ASM.
;
        .386c

        ASSUME  cs:cseg,ds:dseg,ss:sseg

;
; We give the segments a use type of USE32 (the
; default for the 80386).
```

```
;
data     SEGMENT PARA PUBLIC USE32 'DATA'

hellomsg DB      'Hello world.....',0Dh,0Ah,'$'

data     ENDS

sseg     SEGMENT PARA STACK USE32 'STACK'

         DB      1000 DUP (?)

sseg     ENDS

cseg     SEGMENT PARA PUBLIC USE32 'CODE'

print    PROC    NEAR

;
; currently, programs linked by 386|LINK to
; execute in protected mode are set up with a
; single segment, and 386|DOS-Extender initializes
; all segment registers to point to it.  There is
; therefore no need to initialize DS in a
; protected mode program.
;
; Although it is unnecessary, we use the 32 bit
; registers and a protected mode instruction here
; to load the MS-DOS function code into AH.
;
        mov    al,09h   ; load function code into AL
        xor    ebx,ebx ; clear EBX to shift in 0's
        shld   eax,ebx,8 ; shift code into AH
        mov    dx,OFFSET hellomsg
        int    21h      ; call MS-DOS

        mov    ah,4ch
        int    21h      ; terminate process

print   ENDP

cseg    ENDS

        END    print
```

As noted above, there is currently no version of MS-DOS available which supports protected mode operation on the 80386. Phar Lap Software

provides a product, called the 386 | DOS-Extender, which allows programs written to execute in protected mode to execute under the control of MS-DOS, with full availability of MS-DOS system services. Please see the *386 | DOS-Extender Reference Manual* for full details.

The 80286 processor also has a protected mode of operation. It is similar to the 80386 protected mode: There is a GDT and an LDT, and the segment selectors are the same. However, with the 80286, the segment offset is still only 16 bits wide. The segment base address and the linear and physical addresses formed from the segment selector and offset are 24 bits wide, giving an address space of 16 megabytes in 80286 protected mode.

As part of Intel's policy of upward compatibility in the 8086 family, the 80386 provides a way of executing programs written for 80286 protected mode without modification. They provide the concept of a use type (please see section 3.7.3) on a segment. A segment is designated as either a USE16 or a USE32 segment and is identified as such in the descriptor table entry for the segment. USE32 segments are the default and operate as described above. USE16 segments are provided for compatibility with the 80286. Offsets within USE16 segments are 16 bits wide, so the maximum segment size is 64 KB. Instruction operands are assumed to be 16 bits wide, instead of the default of 32 bits wide in USE32 segments. Stack segments with the USE16 attribute use the SP register instead of the ESP register on PUSHs and POPs. It is still possible to use the 32 bit wide register set and manipulate 32 bit wide memory operands from USE16 segments, but instructions that do so must be prefixed by override bytes. 386 | ASM automatically generates operand size and address size override bytes as necessary in USE16 segments.

In general, USE16 segments should never be used with programs written for 80386 protected mode. They should be used only for programs originally written for 80286 protected mode and then ported to the 80386. Unfortunately, it is possible to write programs that mix the use of USE16 and USE32 segments. Although this is not advised, if you do so, there are certain rules to follow for control transfers between USE16 and USE32 segments. Please see Appendix J for details.

11.3 Using the 80386 in Real Mode

Even when the 80386 is executing a program in real mode, it is possible to use the 32 bit register set and the new instructions that were added for the 80386. Access to the 32 bit registers and addressing modes is obtained by prefixing instructions with an override byte. The new instructions are automatically recognized and executed by the processor, even when it is executing in real mode. Note that there is no way to get around the segment size limit of 64 KB imposed in real mode.

To force 386|ASM to assemble the 80386 instructions, the program must be assembled with the 80386 as the target machine (the default). To force override bytes to be generated for 32 bit instruction operands, the segment must be designated a USE16 segment. The program is linked with the 8086 as the target machine, as usual for a real mode program. Thus, to assemble and link a program for execution in real mode, but still use the 32 bit register set and the new instruction set, enter the following commands:

```
386asm hello
386link hello -8086
```

We again modify our example program to demonstrate how the 32 bit registers, etc., can be accessed from a program that executes in real mode. The only difference between this program and our sample protected mode program is that the segments are now USE16 segments, so that code that can be executed in real mode is generated.

```
        TITLE   hello.asm

        .386c   ; assemble for protected mode

        ASSUME  cs:cseg,ds:dseg,ss:sseg

;
; We give the segments a use type of USE16, so
; that the assembler will generate override bytes
; to allow us to access 32 bit operands.
;
data    SEGMENT PARA PUBLIC USE16 'DATA'
```

```
hellomsg DB     'Hello world.....',0Dh,0Ah,'$'

data     ENDS

sseg     SEGMENT PARA STACK USE16 'STACK'

         DB      1000 DUP (?)

sseg     ENDS

cseg     SEGMENT PARA PUBLIC USE16 'CODE'

print    PROC    NEAR

         mov     ax,dseg
         mov     ds,ax            ; Init DS

;
; As in the protected mode program, we use 32 bit
; registers and the new instruction SHLD.  However,
; the code generated will be different;  since we
; are in a USE16 segment, we will get operand size
; override bytes for the 32 bit operands.
;
         mov     al,09h  ; load function code into AL
         xor     ebx,ebx ; clear EBX to shift in 0's
         shld    eax,ebx,8 ; shift code into AH
         mov     dx,OFFSET hellomsg
         int     21h     ; call MS-DOS

         mov     ah,4ch
         int     21h     ; terminate process

print    ENDP

cseg     ENDS

         END     print
```

This page intentionally left blank.

386 | ASM Command Line Switches

386ASM	file1,file2,...,filen	Input file names
	-LIST file	Listing file name
	-NOLIST	Suppress listing file
	-ERRORLIST file	Produce error listing file
	-OBJECT file	Object file name
	-NOOBJECT	Suppress object file
	-8086 -80186 -80286 -80386	Select target CPU
	-80286P -80386P	Select CPU with protected instructions
	-NO87 -8087 -80287 -80387	Select target numeric coprocessor
	-INCLUDE path	Set INCLUDE directive path
	-DEFINE name[=string]	Define a text symbol
	-NOSYM	Suppress symbol table in listing
	-NODELETE	Produce object file even if errors
	-ONECASE -TWOCASE	Set case of symbols
	-FULLWARN	Enable extra error checking

This page intentionally left blank.

Error Messages

If an error occurs during assembly of a program, 386 | ASM displays an error message on the screen which identifies the source code file in which the error occurred, the statement line number within the file, an error number identifying the error, and a message describing the error. The same message is also written to the .LST file.

Assembler errors are divided into three categories: (1) Error numbers less than 1000 are used to identify warning errors. Warning errors are errors that do not necessarily cause the assembler to generate invalid object code for the statement in error. (2) Error numbers greater than 1000 and less than 2000 are used to identify severe errors. Severe errors are errors which cause the assembler to be unable to generate correct object code for the statement. (3) Error numbers greater than 2000 are fatal errors. Fatal errors cause the assembler to terminate immediately, without processing the rest of the source code file. Fatal errors are only written to the listing file if they occur on pass 2 of the assembly.

The three sections below list the warning errors, severe errors and fatal errors that occur in 386 | ASM. Probable causes and possible solutions for the errors are also listed. Errors are listed in numerical order.

B.1 Warning Errors

`WARNING 1:` `Syntax error - name ignored by directive`

Cause: A user-defined name was given to a directive that does not use names.

```
WARNING 2:    Can't close listing file: filename
WARNING 3:    Can't close object file: filename
WARNING 4:    Can't close error list file: filename
WARNING 5:    Can't close cross-reference file:    filename
```

Cause:

1. The disk is full.
2. MS-DOS resident code has been corrupted.

Solution:

1. Delete files to free up space on the disk.
2. Reboot the system.

```
WARNING 6:    Can't close source file: filename
WARNING 7:    Can't close include file: filename
```

Cause: MS-DOS resident code has been corrupted.

Solution: Reboot the system.

```
WARNING 8:    Source line too long - truncated
```

Cause: The source line exceeded 132 characters and was truncated to 132 characters.

Solution: Shorten the source code line.

```
WARNING 9:    at character number n:  Illegal printing
              character - ignored
```

Cause: A printing character which is not part of the character set recognized by the assembler was encountered and was treated as a space.

WARNING 10: `at character number n: Illegal non-printing character - ignored`

Cause: A non-printing character other than a space, carriage return, line feed, horizontal tab, or form feed was encountered and was treated as a space.

WARNING 11: `at character number n: character used in illegal context - ignored`

Cause: A character was used in an illegal context and was treated as a space.

WARNING 13: `String not terminated`

Cause: The end quote on a string constant is missing.

WARNING 14: `Missing ENDIF for conditional block`

Cause: The ENDIF required to terminate a conditional block is missing.

WARNING 15: `Missing ENDM for macro definition`

Cause: The ENDM required to terminate a macro definition or a repeat block is missing. This caused all the rest of the lines in the source file to be treated as part of the macro definition.

`WARNING 16:` `Integer overflow`

Cause: A constant or constant expression is too large for a particular data type.

Solution: Change the data type or give the constant a smaller value.

`WARNING 17:` `Floating point overflow`

Cause: A floating point constant is too large for a particular data type.

Solution: Change the data type or give the constant a smaller value.

`WARNING 18:` `Floating point underflow`

Cause: A floating point constant is too small for a particular data type.

Solution: Change the data type or give the constant a larger value.

`WARNING 19:` `Decimal point following hexadecimal digits`

Cause: A decimal point may only be used with decimal digits.

`WARNING 20:` `Macro purged or redefined within itself: macroname`

Cause: A macro purged or redefined itself. This is legal for compatibility with Microsoft, but it is a somewhat dubious practice.

Solution: Either ignore the warning, or fix the macro definition.

`WARNING 21:`	`Digits too large for radix - base 16 assumed`
Cause:	One or more digits in a constant were too large for the current default radix, or for the radix specified with a radix designator character at the end of the constant.
Solution:	Fix the number or add a radix designator at the end of the constant.

`WARNING 22:`	`Unmatched ENDM`
Cause:	An ENDM was encountered for which there is no matching MACRO, IRP, IRPC, or REPT.

`WARNING 23:`	`Extra characters on line`
Cause:	After all the operands for a directive or instruction were processed, there were still more characters on the source line.

`WARNING 24:`	`Illegal use of symbol in expressions: symbol`
Cause:	Either a user-defined symbol or an assembler reserved word was used illegally in an expression.

`WARNING 25:`	`Divide by zero`
Cause:	A divide by zero was attempted in an expression.

`WARNING 26:` `Reference to multiply defined symbol: symbolname`

Cause: The statement references a symbol that is defined more than once elsewhere in the source code file.

Solution: Remove all but one of the symbol definitions.

`WARNING 27:` `No IF to match ELSE or ENDIF`

Cause: An ELSE or ENDIF statement was encountered when there was no active conditional block.

`WARNING 28:` `Two ELSEs in a row`

Cause: Two ELSE directives were encountered in succession with no ENDIF or IF statement between them.

`WARNING 29:` `Floating point denormal value generated`

Cause: The floating point number is so small it is a denormal value in the specified data type.

`WARNING 30:` `Constant definition will not retain its data type`

Cause: An absolute expression which followed an equal sign (=) directive had a data type which was ignored. The symbol name which preceded the directive was entered into the symbol table as an absolute constant without a data type.

Solution: If the absolute constant does not need to have a data type associated with it, remove the data type specifier from the expression. If the data type is required, the constant can be entered in the symbol table as a text substitution by using the EQU directive instead.

`WARNING 31:` `Instruction shortened, NOPs inserted`

Cause: A forward reference in the instruction caused more space to be allocated on pass one than was necessary on pass two. The unused space was padded out with NOP instructions. (This error is only reported when the -FULLWARN switch is used.)

Solution: Remove the forward reference that caused the warning, or give the assembler more information about the forward-referenced symbol by using the PTR and/or segment override operators.

`WARNING 32:` `Default segment register has no assumption`

Cause: The indirect address modes imply a segment register which is used as a base when the effective address of the operand is calculated. If the segment register implied by the address mode has no assumption, 386 | ASM will flag a WARNING error to inform the programmer that a segment register which has no assumption has been used.

Solution: Use the ASSUME directive to inform the assembler what value the segment register will have when the instruction is executed.

`WARNING 33:` `Value combined with segment selector ignored`

Cause: A segment selector value (obtained by using a segment name or the SEG expression operator) was combined with a constant value. When the program is loaded for execution, the value will be replaced with the segment selector value, and the constant value will be discarded.

B.2 Severe Errors

`ERROR 1002:` `Syntax error`

Cause: There was a syntax error in the statement.

`ERROR 1003:` `Syntax error - instruction expected`

Cause: Something other than an instruction followed an instruction label.

`ERROR 1004:` `Syntax error - directive expected`

Cause: Something other than a directive followed a user–defined symbol.

`ERROR 1005:` `Syntax error - missing argument`

Cause: A required argument for a directive is missing.

`ERROR 1006:` `Can't open include file: filename`

Cause:
1. Invalid file name.
2. Access denied to the file.

Solution:
1. Correct the spelling of the file name.
2. Clear up the access problem.

`ERROR 1007:` `Can't open include file, path too long: filename`

Cause: The file name specified in the INCLUDE statement plus the path name specified in the command line together added up to more than 255 characters.

Solution: Move the include file so that the path name can be made shorter.

`ERROR 1008:` `Syntax error - missing name for   directive`

Cause: A directive that requires a user-defined name is missing the name.

`ERROR 1009:` `Illegal statement - no current segment`

Cause: A statement that is only legal within a segment was encountered when no segment was open. All instructions and many directives must be inside a segment.

`ERROR 1010:` `Symbol already a reserved word: symbolname`

Cause: Attempt to create a user-defined symbol with the same name as one of the assembler reserved words.

`ERROR 1011:` `Symbol already different kind: symbolname`

Cause: Attempt to redefine a previously defined symbol.

`ERROR 1012:` `Segment parameters are changed`

Cause: Attempt to re-open a segment with different segment attributes than those used for its definition.

`ERROR 1013:` `Count for DUP operator cannot be relocatable`

Cause: Only constant values may be used for the count parameter with the DUP operator.

`ERROR 1014:` `Improper segment attribute specifier.`

Cause: One or more of the keywords following a segment directive is not a valid segment attribute specifier. It must be an align type, combine type, access type or use attribute keyword (e.g., BYTE, PUBLIC, ER, USE16).

`ERROR 1015:` `Block nesting error`

Cause: An attempt was made to re-open a segment that is already open, or an ENDS directive was encountered for a segment or structure which is not currently open.

`ERROR 1016:` `Open procedure: procname`

Cause: The procedure named *procname* was opened with the PROC directive but never closed with a corresponding ENDP directive.

`ERROR 1017:` `Undefined symbol: symbolname`

Cause: A symbol that was never defined was referenced.

`ERROR 1018:` `Open segment: segname`

Cause: The segment named *segname* was opened with the SEGMENT directive but never closed with a corresponding ENDS directive.

`ERROR 1019:` `Symbol multiply defined: symbolname`

Cause: Attempt to redefine a symbol that has been previously defined.

`ERROR 1020:` `Segment already in a different group: segname`

Cause: Attempt to put the segment *segname* into more than one group.

Solution: Remove *segname* from all group definitions except one.

`ERROR 1021:` `Syntax error - invalid expression`

Cause: The expression does not obey the rules for forming a legal expression.

Solution: Check the operator precedence table and look for typing mistakes.

`ERROR 1022:` `DUP operator requires count on left`

Cause: An absolute count is required to the left of the DUP operator in data declaration statements.

`ERROR 1023:` `DUP operator nested too deeply`

Cause: DUP operators were nested greater than 17 levels deep in a data declaration directive.

Solution: Find a simpler way to declare the data.

`ERROR 1024:` `Unknown symbol type specifier`

Cause: A directive which requires a symbol type keyword (e.g., BYTE, NEAR) has not found one.

`ERROR 1025:` `Type illegal in context`

Cause: The symbol type keyword used with the directive is not valid given the context of the statement.

`ERROR 1026:` `No or unreachable CS`

Cause: A NEAR procedure has been defined when the CS register is assumed to be pointing at a different segment.

Solution: Use the ASSUME directive to tell the assembler that CS will be pointing at the current segment before defining a NEAR procedure.

`ERROR 1027:` `Procedure not open: procname`

Cause: An ENDP directive was encountered for a procedure which is not currently open.

Solution: Remove the useless ENDP directive.

`ERROR 1028:` `Procedure nesting error, closing: procname`

Cause: An ENDP directive was encountered for a procedure other than the most recently opened one, causing the most recently opened one to be closed.

Solution: Correctly order the PROC/ENDP directives for each procedure.

`ERROR 1029:` `Procedure closed in different segment: procname`

Cause: The named procedure was opened in one segment and closed in another.

`ERROR 1030:` `Constant was expected`

Cause: A relocatable value was used when only a constant value is permitted.

`ERROR 1031:` `Module already has title`

Cause: Two TITLE directives were encountered. Only one is permitted per module.

`ERROR 1032:` `Can't use EVEN on byte segment`

Cause: The EVEN directive was used in a segment with a byte align type. Since the assembler can't know whether the segment will reside on an even or an odd byte boundary, it is unable to decide whether or not to generate a pad byte.

Solution: Change the align type of the segment or remove the directive.

`ERROR 1033:` `Value is out of range`

Cause: The value is too large or too small for the data type either explicitly specified or implied in the statement.

`ERROR 1034:` `Illegal forward reference to symbol: symbolname`

Cause: The forward reference to the specified symbol is illegal, because the assumptions made by the assembler about the symbol on pass one caused less object code to be generated than on pass 2.

Solution: Use operators, such as segment override or PTR, to give the assembler more information about the forward referenced symbol; or move the symbol declaration in front of the statement that references it.

`ERROR 1035:` `Syntax error - bad data type in expression`

Cause: One of the constant data types other than integer (e.g., packed decimal, real number, quadword integer) was used with the wrong data declaration directive (DB, DW, DQ, DP, DT).

`ERROR 1036:` `Symbol already defined external: symbolname`

Cause: An attempt to locally define an external which has already been referenced in the module. An external can only be redefined locally if has not been used yet.

Solution: Remove the local definition or remove all the references above it.

`ERROR 1037:` `Symbol not defined external: symbolname`

Cause: A symbol was declared both public and external in a module and the public definition took precedence.

Solution: Remove the external declaration.

`ERROR 1038:` `Symbol already defined locally: symbolname`

Cause: An attempt to declare as external a symbol which has already been defined locally.

Solution: Remove the EXTRN definition for the symbol.

ERROR 1039: `Local and external definitions differ for: symbolname`

Cause: The symbol data type given in the external declaration for a symbol does not match its local definition.

ERROR 1040: `Symbol type illegal for PUBLIC: symbolname`

Cause: A symbol other than a variable, label, or constant was used in a PUBLIC definition.

ERROR 1041: `Segment override is illegal`

Cause: A segment override is illegal in the context of the statement.

ERROR 1042: `Improper operand type`

Cause: The data type of the operand is illegal for the instruction or directive.

ERROR 1043: `Operand types don't match`

Cause: Two or more operands of an instruction or directive have incompatible data types.

ERROR 1044: `Circular text substitution: symbolname`

Cause: The specified symbol name is a text substitution symbol which, if expanded, would cause a circular text substitution.

`ERROR 1045:` `Multiply defined symbol: symbolname`

Cause: The symbol is defined in more than one place in the source program.

`ERROR 1046:` `Missing )`

Cause: There was a "(" with no matching ")" in an expression.

`ERROR 1047:` `Missing ]`

Cause: There was a "[" with no matching "]" in an expression.

`ERROR 1048:` `Invalid use of relocatable operand`

Cause: A relocatable value was combined in an illegal fashion with an expression operator.

`ERROR 1049:` `Invalid use of segment override`

Cause: The expression attempted to combine two values with overrides to different segments.

`ERROR 1050:` `Incompatible data types in expression`

Cause: The expression attempted to combine two values with different data types.

`ERROR 1051:` `Illegal use of register`

Cause: A register name was used illegally in an expression.

`ERROR 1052:` `Illegal constant value for expression`

Cause: One of the constant data types other than integer (e.g., packed decimal, real number, quadword integer) was used with an operator other than unary + or unary -.

`ERROR 1053:` `Operand must have size`

Cause: The assembler must know the size of the operand(s) to an instruction in order to generate correct object code.

Solution: Specify the size of the operand(s) with the PTR operator.

`ERROR 1054:` `Illegal addressing mode`

Cause: An illegal addressing mode for the instruction was used.

`ERROR 1056:` `Can't reach with segment register`

Cause: A memory reference operand cannot be referenced with any of the currently assumed segment register values.

Solution: Use the ASSUME directive to tell the assembler which segment register will point at the operand's segment at runtime.

`ERROR 1057:` `Illegal scale factor for index register`

Cause: A scale factor other than one, two, four, or eight was used with an index register.

`ERROR 1059:` `ST(n) 8087 register out of range`

Cause: A value outside the range zero–seven was used to identify one of the 8087 floating point stack registers.

`ERROR 1060:` `Text substitution (from EQU) caused buffer overflow - line truncated`

Cause: The source statement contains references to one or more text substitution symbols which, when expanded, caused the assembler's internal buffers to overflow, forcing it to truncate the internal representation of the statement.

Solution: Break up the statement into two or more statements.

`ERROR 1061:` `Near JMP/CALL to different CS`

Cause: A near JMP or CALL was attempted to a label in another segment.

`ERROR 1062:` `AT combine value must be last on line`

Cause: No segment attribute may be given following the AT combine type specifier with the SEGMENT directive.

`ERROR 1063:` `Relative value out of range`

Cause: A PC-relative value is greater than 127 or less than -128.

`ERROR 1064:` `Wrong number of digits in hexadecimal real`

Cause: A hexadecimal encoded real number does not have either 8, 16 or 20 digits.

`ERROR 1065:` `Illegal packed decimal number`

Cause: A packed decimal number has more than 18 decimal digits, or has digits (i.e., A–F) that are too large for base 10.

`ERROR 1066:` `Syntax error - no expression following % operator`

Cause: The % operator in a macro invocation statement was not followed by a valid expression.

`ERROR 1067:` `Missing END directive added by assembler`

Cause: The source file did not contain an END statement. The assembler inserted an END statement following the last line in the last source file of the assembly.

`ERROR 1068:` `Statement only valid inside macro`

Cause: The LOCAL or EXITM directives were encountered outside a macro definition.

`ERROR 1069:` `SHORT operator only valid for label`

Cause: The SHORT expression operator was used with a value other than an instruction label.

`ERROR 1070:` `Missing >`

Cause: A "<" with no matching ">" was encountered.

`ERROR 1071:` `Cannot override ES register`

Cause: The destination operand of a string must be compatible with the ES register.

Solution: Use the ASSUME directive or segment override operator to make sure the operand can be reached with ES.

`ERROR 1072:` `Long parameter caused line truncation in macro expansion`

Cause: Parameter substitution in a macro expansion caused a source code line longer than 132 characters to be generated. The source line was truncated to 132 characters.

`ERROR 1073:` `IF directives nested too deeply`

Cause: IF directives were nested more than 255 levels deep.

`ERROR 1074:` `Macros nested too deeply`

Cause: Macro expansions were nested more than 65,535 levels deep.

`ERROR 1076:` `Missing ENDS for structure definition`

Cause: The ENDS required to terminate a structure definition block is missing.

`ERROR 1077:` `Forced error`

Cause: ERROR was forced by .ERR directive.

`ERROR 1078:` `Forced error - expression equals 0`

Cause: ERROR was forced by .ERRE directive.

`ERROR 1079:` `Forced error - expression not equal to 0`

Cause: ERROR was forced by .ERRNE directive.

`ERROR 1080:` `Forced error - symbol defined`

Cause: ERROR was forced by .ERRDEF directive.

`ERROR 1081:` `Forced error - symbol not defined`

Cause: ERROR was forced by .ERRNDEF directive.

`ERROR 1082:` `Forced error - string blank`

Cause: ERROR was forced by .ERRB directive.

`ERROR 1083:` `Forced error - string not blank`

Cause: ERROR was forced by .ERRNB directive.

`ERROR 1084:` `Forced error - string identical`

Cause: ERROR was forced by .ERRIDN directive.

`ERROR 1085:` `Forced error - strings different`

Cause: ERROR was forced by .ERRDIF directive.

ERROR 1086: `Cannot change target CPU after a segment definition`

Cause: Attempt to use .8086, .186, .286, .286c, .286p, .386, .386c, .386p, or .PROT after a segment was opened.

ERROR 1087: `Segment access types must form a compatible set`

Cause: Attempt to define a segment with access types which are not combinable. For example, ER and RW.

ERROR 1088: `Module already has a name`

Cause: Two NAME directives were encountered in the same source program.

ERROR 1089: `Statement illegal inside structure definition`

Cause: A structure definition block is currently open, and the statement is not legal within a structure definition. Only data declaration statements and comments are legal within structure definitions.

ERROR 1090: `Field cannot be overridden`

Cause: An attempt was made to initialize a structure field which cannot be initialized when creating a structure instance. Any field containing multiple values cannot be initialized.

ERROR 1091: `Override value is too long`

Cause: A constant string used to initialize a structure field is too long to fit in the field.

`ERROR 1092:` `Override with DUP is illegal`

Cause: An initializer for a structure or record field had a DUP operator in the expression. The DUP operator is illegal in expressions for initializers.

`ERROR 1093:` `More values than defined fields`

Cause: There are more values inside the angle brackets for a structure or record declaration than there are fields in the structure or record.

`ERROR 1094:` `Override with string is legal on byte fields only`

Cause: An attempt was made to use a constant string as an initializer for a field which does not have a data type of BYTE.

`ERROR 1095:` `Use of structure requires initializer list`

Cause: An attempt was made to create an instance of a structure without specifying an initializer list. The angle brackets (<>) must be present, even if the list is empty.

`ERROR 1096:` `Field width specifier must be absolute constant`

Cause: The width specifier for a record field being defined is required to be an absolute constant.

`ERROR 1097:` `Field value must be absolute constant`

Cause: The value for a record field must be an absolute constant.

`ERROR 1098:` `Default field value is too large for field width`

Cause: The default value for a field in the record being created is too large to fit in the width specified for the field.

`ERROR 1099:` `Initial field value is too large for field width`

Cause The initializer for a record field is too large to fit in the field that is being initialized.

`ERROR 1100:` `Total size of record fields is too large`

Cause: The total size of the fields in the record being created is larger than 32 bits (or larger than 16 bits and the target CPU is not an 80386).

`ERROR 1101:` `Use of record requires initializer list`

Cause: An attempt was made to create an instance of a record without specifying an initializer list. The angle brackets (<>) must be present, even if the list is empty.

`ERROR 1102:` `String illegal for record field override`

Cause: An attempt was made to use a constant string as an initializer for a field in a record instance being created.

`ERROR 1103:` `Local symbol not allowed with this directive`

Cause: A symbol local to a procedure block (a symbol that begins with a "#" character) can not be defined with this directive. Only the data declaration directives, EQU, LABEL, and =, can be used to create local symbols.

`ERROR 1104:` `Local symbol must be defined inside a procedure`

Cause: A local symbol (a symbol beginning with a "#" character) can only be defined inside a procedure block.

`ERROR 1105:` `Align directive requires a power of two`

Cause: The operand to the align directive must be a power of 2, or the location counter will not be changed.

`ERROR 1106:` `Descriptor table register requires PWORD operand`

Cause: Instructions which operate on the descriptor table registers (IDTR and GDTR) require their memory reference operand to have a data type of PWORD.

Solution: The PTR operator can be used to coerce the data type of memory reference operand to PWORD if necessary.

`ERROR 1107:` `Location counter has wrapped`

Cause: The location counter in the currently open segment has overflowed and wrapped around. This will cause the initialized data at the beginning of the segment to be destroyed and make the size of the segment incorrect at link time.

`ERROR 1108:` `Count for DUP operator cannot be a forward reference`

Cause: The count parameter used with the DUP operator must be a constant value, and cannot include a reference to a symbol defined later in the source file.

`ERROR 1109: Relocatable ORG value must lie in current segment`

Cause: The value for the ORG directive was relocated relative to a different segment than the one currently being assembled. 386|ASM requires the operand to ORG to be absolute or relocated relative to the currently open segment.

B.3 Fatal Errors

`FATAL ERROR 2001: System error`

Cause: A bug in 386|ASM. Save a copy of the source code file(s) which caused the error, and call Phar Lap.

```
FATAL ERROR 2002: Can't create listing file: filename
FATAL ERROR 2003: Can't create object file: filename
FATAL ERROR 2004: Can't create error list file: filename
FATAL ERROR 2005: Can't create cross-reference file: filename
```

Cause:
1. You have misspelled the name of the file or its directory.
2. The directory in which the file is to be placed does not exist, or you do not have write access to the directory.
3. The disk is full.

Solution:
1. Correctly spell the file or directory name.
2. Clear up the access problem.
3. Discard unused files to free up space on the disk.

```
FATAL ERROR 2006: Error writing to listing file: filename
FATAL ERROR 2007: Error writing to object file: filename
FATAL ERROR 2008: Error writing to error list file: filename
FATAL ERROR 2009: Error writing to cross-reference file:
                  filename
```

Cause: The disk is full.

Solution: Discard unused files to free up space on the disk.

```
FATAL ERROR 2010: Can't open source code file: filename
```

Cause:
1. You have misspelled the name of the file or its directory.
2. You do not have read access to the file or directory.

Solution:
1. Correctly spell the file or directory name.
2. Clear up the access problem.

```
FATAL ERROR 2011: Error reading source code file: filename
```

Cause: The file is probably damaged.

Solution: Recreate the file, or restore it from a backup copy.

```
FATAL ERROR 2012: Out of memory
```

Cause: 386 | ASM has run out of space to keep the symbol table in memory.

Solution: Break up the source code file into two or more files to be assembled separately and combined with 386 | LINK.

```
FATAL ERROR 2013: Symbol changed offset on pass 2
FATAL ERROR 2014: Procedure changed size on pass 2
FATAL ERROR 2015: Segment changing size on pass 2
```

Cause: A bug in 386 | ASM. Save a copy of the source code file(s) that caused the error, and call Phar Lap.

```
FATAL ERROR 2016: Illegal symbol name defined on
                  command line: symbolname
```

Cause: The symbol name defined on the command line with the -DEFINE switch does not follow the syntax rules for legal symbol name construction.

This page intentionally left blank.

Appendix C

Syntactical Elements

C.1 Character Set

386 | ASM recognizes the following character set:

```
A-Z
a-z
0-9
+ - * /
() [] <>
' "
_ @ ? $ #
, ; . : =
% ! &
```

space carriage return line feed horizontal tab

When 386 | ASM encounters any character not in this set, it treats it as a space.

C.2 Statements

386 | ASM processes source code files one line at a time, where a line is terminated by a line feed character or or by a carriage return/line feed character pair. Source code lines may be no longer than 132 characters in length.

386 | ASM parses each source code line to break it up into tokens. Tokens fall into one of the following categories: delimiters, assembler reserved names, identifiers (user-defined names), constant numeric values, and string constants. Delimiters are always a single source character. All other tokens may be one or more source characters in length and are terminated by any character which does not follow the syntax rules for the token type.

A statement must be on a single line of the source file and takes the general form:

```
{identifier} mnemonic {operand,...}    {;comment}
```

where items in brackets are optional and *mnemonic* is an instruction mnemonic or a directive name.

C.3 Assembler Reserved Names

A reserved name is a name that has a special meaning to 386|ASM. Reserved names may not be redefined. Reserved names include instructions, directives, register names, and expression operators, all of which are listed in separate appendices in this manual. In addition, the names listed in Table C-1 are reserved names. All upper and lower case combinations of these names are treated as the same name, regardless of whether case-sensitive assembly is enabled.

TABLE C-1
MISCELLANEOUS RESERVED NAMES

Name	80836 only	Description
ABS	no	Data type used with EXTRN directive
BYTE	no	Variable data type
DWORD	no	Variable data type
DUP	no	Operator used with data declarations to specify replication of values
EO	yes	Segment access type attribute
ER	yes	Segment access type attribute
FAR	no	Label data type
FWORD	yes	Variable data type
NEAR	no	Label data type
NOTHING	no	Keyword for ASSUME directive

(cont.)

TABLE C-1, CONTINUED		
Name	80836 only	Description
PARA	no	Segment align type
PWORD	yes	Variable data type
QWORD	no	Variable data type
RO	yes	Segment access type attribute
RW	yes	Segment access type attribute
TBYTE	no	Variable data type
USE16	yes	Segment use attribute
USE32	yes	Segment use attribute
WORD	no	Variable data type

C.4 Identifiers

Identifiers are user-defined names. Identifiers may be any combination of the following characters:

```
A-Z
a-z
0-9
_ ? $ @
```

The identifier may not begin with a number. Only the first 31 characters in an identifier are significant; all others are ignored. Unless case sensitivity is enabled with a command line switch, upper and lower case versions of the same identifier are considered to be identical by 386|ASM.

In addition, local identifiers may be defined within a procedure block by beginning the identifier name with the pound character (#). Local identifiers are recognized only within the procedure block in which they are defined.

C.5 Delimiters

Delimiters are one-character tokens with a particular meaning, sometimes context sensitive, to 386 | ASM. The delimiters recognized by 386 | ASM are listed in Table C-2.

TABLE C-2
DELIMITERS

Delimiter	Description
+ - * / . : () []	Expression operators
<> & ! %	Macro operators
;	Beginning of comment portion of a statement
,	Separates operands of an instruction or directive
$	Represents current value of location counter within the current segment
' "	Used to enclose string constants

C.6 Constants

Integers:

digits{radix_specifier}

Integer numbers are a string of one or more digits in the default radix (which may be set by use of the .RADIX directive). The A-F digits used to enter hexadecimal numbers may be given in upper or lower case. A radix specifier (Table C-3) may be appended to the digit string to override the default radix. Note that hexadecimal numbers must always start with a decimal digit (0-9); tokens that begin with a letter are treated as identifiers. If necessary, put a leading 0 on a hexadecimal number to identify it as a number.

TABLE C-3
RADIX SPECIFIERS

Radix Specifier	Radix
B	binary (base 2)
Q	octal (base 8)
O	octal (base 8)
D	decimal (base 10)
H	hexadecimal (base 16)

Real Numbers:

digits.{*digits*}{E{+|-}*digits*}

A real number is composed of a required integer part, a required decimal point, an optional fractional part, and an optional exponent. All digits must be decimal (0-9). Real numbers may only be used with the DD, DQ, and DT directives. Please see the appendix on data types for the range of real numbers allowed for each directive.

Encoded Real Numbers:

*digits*R

Encoded real numbers are entered as an eight digit (for DD), 16 digit (for DQ), or 20 digit (for DT) hexadecimal number followed by the real number specifier R. If the number has a leading 0, the number of its digits may be increased by one.

Packed Decimal Numbers:

digits

A packed decimal number is a decimal number stored in 8087 packed decimal format. A packed decimal number is created by the DT directive when the default radix is decimal and no radix specifier is used at the end

of the number. The maximum number of digits in a packed decimal number is 18.

C.7 String Constants

String constants are a string of one or more printable ASCII characters, non-printable ASCII characters other than the NUL (0) character, or non-ASCII characters (characters with bit 7 set) enclosed in double quotation marks (") or right single quotation marks (').

To encode the quotation mark that encloses the string as part of the string, the quotation mark character must be typed twice. For example, "a""b""c" encodes the string a"b"c.

Strings of up to eight characters in length may also be used as integer values provided they will not overflow the specified data type. The ASCII value of each character in the string is stored as one byte in the integer, with the rightmost character in the string stored as the least significant byte in the integer.

80386 Instruction Set

TABLE D-1
80386 INSTRUCTION SET

Instruction	Processor Required	Syntax	Description
AAA	8086	AAA	ASCII adjust AL for addition
AAD	8086	AAD	ASCII adjust AL for division
AAM	8086	AAM	ASCII adjust AL for multiplication
AAS	8086	AAS	ASCII adjust AL for subtraction
ADC	8086	ADC *accum, immed*	Add immediate with carry to accumulator
	8086	ADC *r/m, immed*	Add immediate with carry to operand
	8086	ADC *r/m, reg*	Add register with carry to operand
	8086	ADC *reg, r/m*	Add operand, carry to register
ADD	8086	ADD *accum, immed*	Add immediate to accumulator
	8086	ADD *r/m, immed*	Add immediate to operand
	8086	ADD *reg, r/m*	Add operand to register

(cont.)

TABLE D-1, CONTINUED

Instruction	Processor Required	Syntax	Description
AND	8086	AND *accum, immed*	AND immediate with accumulator
	8086	AND *r/m, immed*	AND immediate with operand
	8086	AND *r/m, reg*	AND register with operand
	8086	AND *reg, r/m*	AND operand with register
ARPL	80286 P	ARPL *mem, reg*	Adjust requested privilege level
BOUND	80186	BOUND *reg, mem*	Detect value out of range
BSF	80386	BSF *reg, r/m*	Find least significant one bit
BSR	80386	BSR *reg, r/m*	Find most significant one bit
BT	80386	BT *r/m, reg*	Test bit indexed by register
	80386	BT *r/m, immed*	Test bit indexed by immediate
BTC	80386	BTC *r/m, reg*	Test and complement bit indexed by register
	80386	BTC *r/m, immed*	Test and complement bit indexed by immediate
BTR	80386	BTR *r/m, reg*	Test and clear bit indexed by register
	80386	BTR *r/m, immed*	Test and clear bit indexed by immediate
BTS	80386	BTS *r/m, reg*	Test and set bit indexed by register
	80386	BTS *r/m, immed*	Test and set bit indexed by immediate

(cont.)

TABLE D-1, CONTINUED

Instruction	Processor Required	Syntax	Description
CALL	8086	CALL *label*	Call instruction at label
	8086	CALL *r/m*	Call instruction indirect
CBW	8086	CBW	Sign extend AL to AX
CDQ	80386	CDQ	Sign extend EAX to EDX:EAX
CLC	8086	CLC	Clear carry flag
CLD	8086	CLD	Clear direction flag
CLI	8086	CLI	Clear interrupt flag
CLTS	80286 P	CLTS	Clear task-switched flag
CMC	8086	CMC	Complement carry flag
CMP	8086	CMP *accum, immed*	Compare immediate with accumulator
	8086	CMP *r/m, immed*	Compare immediate with operand
	8086	CMP *r/m, reg*	Compare register with operand
	8086	CMP *reg, r/m*	Compare operand with register
CMPS	8086	CMPS *src, dest*	Compare strings
CMPSB	8086	CMPSB	Compare BYTE strings
CMPSD	8086	CMPSD	Compare DWORD strings
CMPSW	8086	CMPSW	Compare WORD strings
CWD	8086	CWD	Sign extend AX to DX:AX

(cont.)

TABLE D-1, CONTINUED			
Instruction	Processor Required	Syntax	Description
CWDE	80386	CWDE	Sign extend EAX to EDX:EAX
DAA	8086	DAA	Decimal adjust AL for addition
DAS	8086	DAS	Decimal adjust AL for subtraction
DEC	8086	DEC *r/m*	Decrement operand
	8086	DEC *reg*	Decrement full-size register
DIV	8086	DIV *r/m*	Divide accumulator by operand
ENTER	80186	ENTER *imm16, 1mm8*	Enter procedure
ESC	8086	ESC *immed, r/m*	Escape with 6 bit immed and operand
HLT	8086	HLT	Halt
IDIV	8086	IDIV *r/m*	Integer divide accumulator by operand
IMUL	8086	IMUL *r/m*	Integer multiply accum by operand
	80186	IMUL *reg, immed*	Integer multiply register by immediate
	80186	IMUL *reg, r/m, immed*	Operand by immed, result in register
	80386	IMUL *reg, r/m*	Multiply register by operand

(cont.)

TABLE D-1, CONTINUED

Instruction	Processor Required	Syntax	Description
IN	8086	IN *accum, immed*	Input from port (8 bit immediate)
	8086	IN *accum*, DX	Input from port given by DX
INC	8086	INC *r/m*	Increment operand
	8086	INC *reg*	Increment 16 bit register
INS	80186	INS *mem*, DX	Input string from port DX
INSB	80186	INSB	Input BYTE string from port DX
INSD	80386	INSD	Input DWORD from port DX to dest string
INSW	80186	INSW	Input WORD string from port DX
INT 3	8086	INT 3	Software interrupt 3 (breakpoint interrupt)
INT	8086	INT *immed*	Software interrupt 0-255
INTO	8086	INTO	Interrupt on overflow
IRET	8086	IRET	Return from interrupt
IRETD	80386	IRETD	Protected mode return from interrupt
JA	8086	JA *label*	Jump if above
JAE	8086	JAE *label*	Jump if above or equal
JB	8086	JB *label*	Jump if below
JBE	8086	JBE *label*	Jump if below or equal
JC	8086	JC *label*	Jump if carry
			(cont.)

TABLE D-1, CONTINUED			
Instruction	Processor Required	Syntax	Description
JCXZ	8086	JCXZ *label*	Jump if CX register is zero
JE	8086	JE *label*	Jump if equal
JECXZ	80386	JECXZ *label*	Jump if ECX register is zero
JG	8086	JG *label*	Jump if greater
JGE	8086	JGE *label*	Jump if greater or equal
JL	8086	JL *label*	Jump if less than
JLE	8086	JLE *label*	Jump if less than or equal
JMP	8086	JMP *label*	Jump to instruction at label
	8086	JMP *r/m*	Jump to instruction indirect
JNA	8086	JNA *label*	Jump if not above
JNAE	8086	JNAE *label*	Jump if not above or equal
JNB	8086	JNB *label*	Jump if not below
JNBE	8086	JNBE *label*	Jump if not below or equal
JNC	8086	JNC *label*	Jump if not carry
JNE	8086	JNE *label*	Jump if not equal
JNG	8086	JNG *label*	Jump if not greater
JNGE	8086	JNGE *label*	Jump if not greater or equal
JNL	8086	JNL *label*	Jump if not less than
JNLE	8086	JNLE *label*	Jump if not less than or equal
JNO	8086	JNO *label*	Jump if overflow flag is clear
JNP	8086	JNP *label*	Jump if parity flag is clear

(cont.)

TABLE D-1, CONTINUED			
Instruction	Processor Required	Syntax	Description
JNS	8086	JNS *label*	Jump if sign flag is set
JNZ	8086	JNZ *label*	Jump if not zero
JO	8086	JO *label*	Jump if overflow flag is set
JP	8086	JP *label*	Jump if parity flag is set
JPE	8086	JPE *r/m*	Jump if parity even
JPO	8086	JPO *label*	Jump if parity odd
JS	8086	JS *label*	Jump if sign flag is set
JZ	8086	JZ *label*	Jump if zero flag is set
LAHF	8086	LAHF	Load AH with flags
LAR	80286 P	LAR *reg, mem*	Load access rights
LDS	8086	LDS *reg, r/m*	Load FAR pointer into DS: register
LEA	8086	LEA *r/m*	Load effective address of operand
LEAVE	80186	LEAVE	Leave procedure
LES	8086	LES *reg, r/m*	Load FAR pointer into ES: register
LFS	80386	LFS *reg, r/m*	Load FAR pointer into FS: register
LGDT	80286 P	LGDT *mem*	Load GDT register from memory (24 bit base)
LGDTE	80386 P	LGDTE *mem*	Load GDT register from memory (32 bit base)
			(cont.)

TABLE D-1, CONTINUED

Instruction	Processor Required	Syntax	Description
LGS	80386	LGS *reg, r/m*	Load far pointer into GS: register
LIDT	80286 P	LIDT *mem*	Load IDT register from memory (24 bit base)
LIDTE	80386 P	LIDTE *mem*	Load IDT register from memory (32 bit base)
LLDT	80286 P	LLDT *mem*	Load local-descriptor table register
LMSW	80286 P	LMSW *mem*	Load machine-status word
LOCK	8086	LOCK	Lock bus
LODS	8086	LODS *src*	Load string element
LODSB	8086	LODSB	Load BYTE from string into AL
LODSD	80386	LODSD	Load DWORD from source string into EAX
LODSW	8086	LODSW	Load WORD from string into AX
LOOP	8086	LOOP *label*	Loop to instruction at label
LOOPE	8086	LOOPE *label*	Loop while equal to instruction at label
LOOPNE	8086	LOOPNE *label*	Loop while not equal to instruction at label
LOOPNZ	8086	LOOPNZ *label*	Loop while not zero to instruction at label
LOOPZ	8086	LOOPZ *label*	Loop while zero to instruction at label

(cont.)

TABLE D-1, CONTINUED

Instruction	Processor Required	Syntax	Description
LSL	80286 P	LSL *reg, mem*	Load segment limit
LSS	80386	LSS *reg, r/m*	Load FAR pointer into SS: register
LTR	80286 P	LTR *mem*	Load task register
MOV	8086	MOV *accum, mem*	Move memory to accumulator
	8086	MOV *mem, accum*	Move accumulator to memory
	8086	MOV *r/m, immed*	Move immediate to operand
	8086	MOV *r/m, reg*	Move register to operand
	8086	MOV *r/m, segreg*	Move segment register to operand
	8086	MOV *reg, immed*	Move immediate to register
	8086	MOV *reg, r/m*	Move operand to register
	8086	MOV *segreg, r/m*	Move operand to segment register
	80386 P	MOV *reg, Creg*	Move control register to general register
	80386 P	MOV *reg, Dreg*	Move debug register to general register
	80386 P	MOV *reg, Treg*	Move test register to general register
	80386 P	MOV *Creg, reg*	Move register to control register
	80386 P	MOV *Dreg, reg*	Move register to debug register
	80386 P	MOV *Treg, reg*	Move register to test register
MOVS	8086	MOVS *dest, src*	Move string

(cont.)

TABLE D-1, CONTINUED

Instruction	Processor Required	Syntax	Description
MOVSB	8086	MOVSB	Move BYTE string
MOVSD	80386	MOVSD	Move DWORD string
MOVSW	8086	MOVSW	Move WORD string
MOVSX	80386	MOVSX *reg, r/m*	Move byte/word operand to word/dword register with zero extend
MOVZX	80386	MOVZX *reg,r/m*	Move byte/word operand to word/dword register with sign extend
MUL	8086	MUL *r/m*	Multiply accumulator by operand
NEG	8086	NEG *r/m*	(2's complement) operand
NOP	8086	NOP	No operation
NOT	8086	NOT *r/m*	(1's complement) operand
OR	8086	OR *accum, immed*	OR immediate with accumulator
	8086	OR *r/m, immed*	OR immediate with operand
	8086	OR *r/m, reg*	OR register with operand
	8086	OR *reg, r/m*	OR operand with register
OUT	8086	OUT DX, *accum*	Output to port given by DX
	8086	OUT *immed, accum*	Output to port (8 bit immediate)
OUTS	80186	OUTS DX, *mem*	Output string to port DX

(cont.)

TABLE D-1, CONTINUED

Instruction	Processor Required	Syntax	Description
OUTSB	80186	OUTSB DX, *mem*	Output BYTE from source string to port DX
	80386	OUTSD	Output DWORD from source string to port DX
OUTSW	80186	OUTSW DX, *mem*	Output WORD from source string to port DX
POP	8086	POP *r/m*	Pop operand from stack
	8086	POP *reg*	Pop register from stack
	8086	POP *segreg*	Pop segment register from stack
POPA	80186	POPA	Pop all 16 bit general registers from stack
POPAD	80386	POPAD	Pop all 32 bit general registers from stack
POPF	8086	POPF	Pop flags from stack
POPFD	80386	POPFD	Pop 32 bit EFLAGS register
PUSH	8086	PUSH *r/m*	Push 16 bit operand on the stack
	8086	PUSH *reg*	Push 16 bit register on the stack
	8086	PUSH *segreg*	Push segreg on the stack
	80186	PUSH *immed*	Push immediate data on the stack
PUSHA	80186	PUSHA	Push all 16 bit general registers on the stack

(cont.)

TABLE D-1, CONTINUED			
Instruction	Processor Required	Syntax	Description
PUSHAD	80386	PUSHAD	Push all 32 bit general registers
PUSHF	8086	PUSHF	Push flags on the stack
PUSHFD	80386	PUSHFD	Push 32 bit EFLAGS register
RCL	8086	RCL r/*m*, 1	Rotate left through carry by one bit
	8086	RCL *r/m* CL	Rotate left through carry by count in CL
	80186	RCL *r/m, immed*	Rotate left through carry by immediate
RCR	8086	RCR *r/m*, 1	Rotate right through carry by one bit
	8086	RCR *r/m*. CL	Rotate right through carry by count in CL
	80186	RCR *r/m, immed*	Rotate right through carry by immediate
REP	8086	REP	Repeat instruction
REPE	8086	REPE	Repeat instruction if equal
REPNE	8086	REPNE	Repeat instruction if not equal
REPNZ	8086	REPNZ	Repeat instruction if not zero
REPZ	8086	REPZ	Repeat instruction if zero
RET	8086	RET [*immed*]	Return, popping optional bytes from stack

(cont.)

TABLE D-1, CONTINUED

Instruction	Processor Required	Syntax	Description
ROL	8086	ROL *r/m*, 1	Rotate left by one bit
	8086	ROL *r/m*, CL	Rotate left by CL
	80186	ROL *r/m*, *immed*	Rotate left by immediate
ROR	8086	ROR *r/m*, 1	Rotate right by one bit
	8086	ROR *r/m*, CL	Rotate right by CL
	80186	ROR *r/m*, *immed*	Rotate right by immediate
SAHF	8086	SAHF	Store AH into flags
SAL	8086	SAL *r/m*, 1	Shift arithmetic left by one bit
	8086	SAL *r/m*, CL	Shift arithmetic left by CL
	80186	SAL *r/m*, *immed*	Shift arithmetic left by immediate
SAR	8086	SAR *r/m*, 1	Shift arithmetic right by one bit
	8086	SAR *r/m*, CL	Shift arithmetic right by CL
	80186	SAR *r/m*, *immed*	Shift arithmetic right by immediate
SBB	8086	SBB *accum*, *immed*	Subtract immediate with borrow
	8086	SBB *r/m*, *immed*	Subtract immediate with borrow
	8086	SBB *r/m*, *reg*	Subtract register with borrow
	8086	SBB *reg*, *r/m*	Subtract operand with borrow
SCAS	8086	SCAS *dest*	Scan string

(cont.)

TABLE D-1, CONTINUED

Instruction	Processor Required	Syntax	Description
SCASB	8086	SCASB	Scan string for BYTE in AL
SCASD	80386	SCASD	Scan string for DWORD in EAX
SCASW	8086	SCASW	Scan string for WORD in AX
SETA	80386	SETA *r/m*	Set byte if above
SETAE	80386	SETAE *r/m*	Set byte if above or equal
SETB	80386	SETB *r/m*	Set byte if below
SETBE	80386	SETBE *r/m*	Set byte if below or equal
SETC	80386	SETC *r/m*	Set byte on carry
SETE	80386	SETE *r/m*	Set byte if equal
SETG	80386	SETG *r/m*	Set byte if greater
SETGE	80386	SETGE *r/m*	Set byte if greater than or equal
SETL	80386	SETL *r/m*	Set byte if less than
SETLE	80386	SETLE *r/m*	Set byte if less than or equal
SETNA	80386	SETNA *r/m*	Set byte if not above
SETNAE	80386	SETNAE *r/m*	Set byte if not above or equal
SETNB	80386	SETNB *r/m*	Set byte if not below
SETNBE	80386	SETNBE *r/m*	Set byte if not below or equal
SETNC	80386	SETNC *r/m*	Set byte on not carry
SETNE	80386	SETNE *r/m*	Set byte if not equal

(cont.)

TABLE D-1, CONTINUED

Instruction	Processor Required	Syntax	Description
SETNG	80386	SETNG *r/m*	Set byte if not greater than
SETNGE	80386	SETNGE *r/m*	Set byte if not greater than or equal
SETNL	80386	SETNL *r/m*	Set byte if not less than
SETNLE	80386	SETNLE *r/m*	Set byte if not less than or equal
SETNO	80386	SETNO *r/m*	Set byte on not overflow
SETNP	80386	SETNP *r/m*	Set byte if parity flag = zero
SETNS	80386	SETNS *r/m*	Set byte on not sign
SETNZ	80386	SETNZ *r/m*	Set byte if not zero
SETO	80386	SETO *r/m*	Set byte on overflow
SETP	80386	SETP *r/m*	Set byte on parity flag
SETPE	80386	SETPE *r/m*	Set byte if parity even
SETPO	80386	SETPO *r/m*	Set byte if parity odd
SETS	80386	SETS *r/m*	Set byte if sign flag=1
SETZ	80386	SETZ *r/m*	Set byte if zero
SGDT	80286 P	SGDT *mem*	Store GDT register in memory (24 bit base)
SGDTE	80386 P	SGDTE *mem*	Store GDT register in memory (32 bit base)

(cont.)

TABLE D-1, CONTINUED

Instruction	Processor Required	Syntax	Description
SHL	8086	SHL *r/m*, 1	Shift left by one bit
	8086	SHL *r/m*, CL	Shift left by CL
	80186	SHL *r/m, immed*	Shift left by immediate
SHLD	80386	SHLD *r/m, reg, cl*	Shift CL bits from reg left into r/m
	80386	SHLD *r/m, reg, immed*	Shift immediate bit count from reg left into r/m
SHR	8086	SHR *r/m*, 1	Shift right by one bit
	8086	SHR *r/m*, CL	Shift right by CL
	80186	SHR *r/m, immed*	Shift right by immediate
SHRD	80386	SHRD *r/m, reg, cl*	Shift CL bits from reg right into r/m
	80386	SHRD *r/m, reg, immed*	Shift immediate bit count right from reg into r/m
SIDT	80286 P	SIDT *mem*	Store IDT register in memory (24 bit base)
SIDTE	80386 P	SIDTE *mem*	Store IDT register in memory (32 bit base)
SLDT	80286 P	SLDT *mem*	Store local–descriptor table register
SMSW	80286 P	SMSW *mem*	Store machine–status word
STC	8086	STC	Set carry flag
STD	8086	STD	Set direction flag

(cont.)

TABLE D-1, CONTINUED			
Instruction	Processor Required	Syntax	Description
STI	8086	STI	Set interrupt flag
STOS	8086	STOD *dest*	Store string
STOSB	8086	STOSB	Store BYTE from AL at destination string
STOSD	80386	STOSD	Store DWORD from EAX at destination string
STOSW	8086	STOSW	Store WORD from AX at destination string
SUB	8086	SUB *accum, immed*	Subtract immediate from accumulator
	8086	SUB *r/m, immed*	Subtract immediate from operand
	8086	SUB *r/m, reg*	Subtract register from operand
	8086	SUB *reg, r/m*	Subtract operand from register
TEST	8086	TEST *accum, immed*	Test immediate bits in accumulator
	8086	TEST *r/m, immed*	Test immediate bits in operand
	8086	TEST *r/m, reg*	Test register bits in operand
	8086	TEST *reg, r/m*	Test operand bits in register
VERR	80286 P	VERR *mem*	Verify read access
VERW	80286 P	VERW *mem*	Verify write access
WAIT	8086	WAIT	Wait
			(cont.)

TABLE D-1, CONTINUED

Instruction	Processor Required	Syntax	Description
XCHG	8086	XCHG *accum, reg*	Exchange accumulator with register
	8086	XCHG *r/m, reg*	Exchange operand with register
	8086	XCHG *r/m, accum*	Exchange register with accumulator
	8086	XCHG *reg, r/m*	Exchange register with operand
XLAT	8086	XLAT *mem*	Translate
XOR	8086	XOR *accum, immed*	XOR immediate with accumulator
	8086	XOR *r/m, immed*	XOR immediate with operand
	8086	XOR *r/m, reg*	XOR register with operand
	8086	XOR *reg, r/m*	XOR operand with register

TABLE D-2
80287 INSTRUCTION SET

Instruction	Processor Required	Syntax	Description
F2XM1	8087	F2XM1	Calculate $2^x - 1$
FABS	8087	FABS	Take absolute value of top of stack
FADD	8087	FADD	Add ST(1) to ST and pop
	8087	FADD *mem*	Add real in memory to ST
	8087	FADD ST, ST(i)	Add ST(i) to ST
	8087	FADD ST(i), ST	Add ST to ST(i) and pop
FADDP	8087	FADDP ST(i), ST	Add ST to ST(i) and pop stack
FBLD	8087	FBLD *mem*	Load 10-byte BCD value on stack
FBSTP	8087	FBSTP *mem*	Store 10-byte BCD value and pop
FCHS	8087	FCHS	Change sign of the top of the stack
FCLEX	8087	FCLEX	Clear exceptions after WAIT
FCOM	8087	FCOM	Compare ST with ST(1)
	8087	FCOM ST	Compare ST with real in memory
	8087	FCOM ST(i)	Compare ST with ST(i)
FCOMP	8087	FCOMP	Compare ST with ST(i) and pop stack
	8087	FCOMP ST	Compare ST with memory and pop
	8087	FCOMP ST(i)	Compare real with stack and pop stack
FCOMPP	8087	FCOMPP	Compare ST with ST(i) and pop stack twice
FDECSTP	8087	FDECSTP	Decrement floating point stack pointer

(cont.)

TABLE D-2, CONTINUED

Instruction	Processor Required	Syntax	Description
FDISI	8087 only	FDISI	Disable interrupts after WAIT (generates no code for 287 target)
FDIV	8087	FDIV	Divide ST(1) by ST and pop
	8087	FDIV *mem*	Divide ST by real in memory
	8087	FDIV ST, ST(i)	Divide ST by ST(i)
	8087	FDIV ST(i), ST	Divide ST(i) by ST
FDIVP	8087	FDIVP ST(i), ST	Divide ST(i) by ST and pop stack
FDIVR	8087	FDIVR	Divide ST by ST(i), leave result in ST(i) and pop
	8087	FDIVR *mem*	Divide memory by ST, leave result in ST
	8087	FDIVR ST, ST(i)	Divide ST(i) by ST, leave result in ST
	8087	FDIVR ST(i), ST	Divide ST by ST(i), leave result in ST(i)
FDIVRP	8087	FDIVRP ST(i), ST	Divide ST by ST(i), leave result in ST(i) and pop
FENI	8087 only	FENI	Enable interrupts after WAIT (generates no code for 287 target)
FFREE	8087	FFREE	Mark ST as free
	8087	FFREE ST	Mark ST as free
	8087	FFREE ST(i)	Mark ST(i) as free
FIADD	8087	FIADD *mem*	Add two or four-byte integer to ST
FICOM	8087	FICOM *mem*	Compare ST with two or four byte integer

(cont.)

TABLE D-2, CONTINUED			
Instruction	Processor Required	Syntax	Description
FICOMP	8087	FICOMP *mem*	Compare ST with two or four byte integer and pop
FIDIV	8087	FIDIV *mem*	Divide ST by two or four byte integer
FIDIVR	8087	FIDIVR *mem*	Divide two or four byte integer by ST, leave result in ST
FILD	8087	FILD *mem*	Push a two, four, or eight byte integer on the floating point stack
FIMUL	8087	FIMUL *mem*	Two or four byte integer multiply
FINCSTP	8087	FINCSTP	Increment floating point stack pointer
FINIT	8087	FINIT	Initialize processor after WAIT
FIST	8087	FIST *mem*	Store ST as a two or four byte integer at *mem*
FISTP	8087	FISTP *mem*	Store ST as a two, four, or eight byte integer and pop
FISUB	8087	FISUB *mem*	Subtract a two or four byte integer from ST
FISUBR	8087	FISUBR *mem*	Subtract ST from two or four byte integer, leave result in ST
FLD	8087	FLD *mem*	Push a four, eight, or ten byte real on the floating point stack
FLDCW	8087	FLDCW *mem*	Load coprocessor control word
FLDENV	8087	FLDENV *mem*	Load coprocessor environment (14 bytes)
FLD1	8087	FLD1	Push 1.0
FLDL2E	8087	FLDL2E	Push $\log_2 e$ on the floating point stack

(cont.)

TABLE D-2, CONTINUED

Instruction	Processor Required	Syntax	Description
FLDL2T	8087	FLDL2T	Push $\log_2 10$ on the floating point stack
FLDLG2	8087	FLDLG2	Push $\log_{10} 2$ on the floating point stack
FLDLN2	8087	FLDLN2	Push $\log_e 2$ on the floating point stack
FLDPI	8087	FLDPI	Push π on top of stack
FLDZ	8087	FLDZ	Load +0.0 on the floating point stack
FMUL	8087	FMUL	Multiply ST(1) by ST and pop
	8087	FMUL *mem*	Multiply ST by real in memory
	8087	FMUL ST, ST(i)	Multiply ST by ST(i)
	8087	FMUL ST(i), ST	Multiply ST(i) by ST
FMULP	8087	FMULP ST(i), ST	Multiply ST(i) by ST and pop
FNCLEX	8087	FNCLEX	Clear exceptions with no WAIT
FNDISI	8087 only	FNDISI	Disable interrupts with no WAIT (generates no code for 287 target)
FNENI	8087 only	FNENI	Enable interrupts with no WAIT (generates no code for 287 target)
FNINIT	8087	FNINIT	Initialize processor with no WAIT
FNOP	8087	FNOP	No operation
FNSAVE	8087	FNSAVE *mem*	Save coprocessor state (94 bytes), no WAIT
FNSTCW	8087	FNSTCW *mem*	Store coprocessor control word with no WAIT
FNSTENV	8087	FNSTENV *mem*	Store coprocessor environment (14 bytes) with no WAIT

(cont.)

TABLE D-2, CONTINUED

Instruction	Processor Required	Syntax	Description
FNSTSW	8087	FNSTSW *mem*	Store coprocessor status word with no WAIT
	80287	FNSTSW AX	Store 80287 status word in AX with no WAIT
FPATAN	8087	FPATAN	Compute arctan (ST(1)/ST), leave result in ST after popping once
FPREM	8087	FPREM	Compute partial modulus of ST mod ST(1)
FPTAN	8087	FPTAN	Compute tan (ST)
FRNDINT	8087	FRNDINT	Round ST to an integer
FRSTOR	8087	FRSTOR *mem*	Restore coprocessor state (94 bytes)
FSAVE	8087	FSAVE *mem*	Save 8087 state (94 bytes) after WAIT
FSCALE	8087	FSCALE	Add ST(1) to exponent of ST
FSETPM	80287	FSETPM	Set 80287 to protected mode
FSQRT	8087	FSQRT	Calculate square root of ST, leave result in ST
FST	8087	FST ST(i)	Store ST in ST(i)
	8087	FST *mem*	Store ST as 4- or 8-byte real at *mem*
FSTCW	8087	FSTCW *mem*	Store coprocessor control word with WAIT
FSTENV	8087	FSTENV *mem*	Store coprocessor environment after WAIT
FSTP	8087	FSTP *mem*	Store ST as a four, eight, or ten byte real and pop

(cont.)

TABLE D-2, CONTINUED

Instruction	Processor Required	Syntax	Description
FSTSW	8087	FSTSW *mem*	Store coprocessor status word after WAIT
	80287	FSTSW AX	Store 80287 status word in AX after WAIT
FSUB	8087	FSUB	Subtract ST from ST(1) and pop
	8087	FSUB *mem*	Subtract real in memory from ST
	8087	FSUB ST, ST(i)	Subtract ST(i) from ST
	8087	FSUB ST(i), ST	Subtract ST from ST(i)
FSUBP	8087	FSUBP ST(i), ST	Subtract ST from ST(i) and pop
FSUBR	8087	FSUBR	Subtract ST(1) from ST, leave result in ST(1) and pop
	8087	FSUBR *mem*	Subtract ST from real in memory, leave result in ST
	8087	FSUBR ST, ST(i)	Subtract ST from ST(i), leave result in ST
	8087	FSUBR ST(i), ST	Subtract ST(i) from ST, leave result in ST(i)
FSUBRP	8087	FSUBRP ST(i), ST	Subtract ST(i) from ST, leave result in ST(i) and pop
FTST	8087	FTST	Compare ST with 0.0
FWAIT	8087	FWAIT	Wait for last coprocessor operation to complete
FXAM	8087	FXAM	Examine ST, set coprocessor condition codes

(cont.)

TABLE D-2, CONTINUED

Instruction	Processor Required	Syntax	Description
FXCH	8087	FXCH	Exchange ST and ST(1)
	8087	FXCH ST(i)	Exchange ST and ST(i)
FXTRACT	8087	FXTRACT	Decompose ST into exponent and significand
FYL2X	8087	FYL2X	Compute Y $\log_2 x$
FYL2P1	8087	FYL2P1	Compute Y $\log_2(x+1)$

Assembler Directives

TABLE E-1
ASSEMBLER DIRECTIVES

Directive Name	Statement Syntax	Description
.186	.186	Enables assembly of 80186 instructions.
.286c .286	.286c .286	Enables assembly of 80286 nonprotected instructions.
.286p	.286p	Enables assembly of all 80286 instructions.
.287	.287	Enables assembly of 80287 instructions.
.386c .386	.386c .386	Enables assembly of 80386 nonprotected instructions. This is the default mode.
.386p	.386p	Enables assembly of all 80386 instructions.
.387	.387	Enables assembly of 80387 instructions.
.8086	.8086	Enables assembly of 8086 instructions.

(cont.)

TABLE E-1, CONTINUED

Directive Name	Statement Syntax	Description
.8087	.8087	Enables assembly of 8087 instructions.
=	*name = expression*	Creates the symbol *name* with the value of *expression*. The symbol will be a constant, variable, or label.
ASSUME	ASSUME *registername:segname*	Indicates to the assembler that *registername* will point at the segment or group named *segname* during the execution of the program. If *segname* is NOTHING, the register is assumed to have an undefined value.
COMMENT	COMMENT *delim text delim*	Causes the assembler to treat all *text* between the two occurrences of *delim* as a comment. *Text* may include carriage returns, so multi-line comments are possible.
.DB	{*name*} DB *value,...*	Allocates and initializes one byte of memory for each *value*. If the optional *name* is present, a variable of type BYTE is created with the specified name.
DW	{*name*} DW *value,...*	Allocates and initializes one word (two bytes) of memory for each *value*. If the optional *name* is present, a variable of type WORD is created with the specified name.

(cont.)

TABLE E-1, CONTINUED

Directive Name	Statement Syntax	Description
DD	{*name*} DD *value*,...	Allocates and initializes a double word (four bytes) of memory for each *value*. If the optional *name* is present, a variable of type DWORD is created with the specified name.
DP DF	{*name*} DP *value*,... {*name*} DF *value*,...	Allocates and initializes three words (six bytes) of memory for each *value*. If the optional *name* is present, a variable of type PWORD is created with the specified name.
DQ	{*name*} DQ *value*,...	Allocates and initializes a quad-word (eight bytes) of memory for each *value*. If the optional *name* is present, a variable of type QWORD is created with the specified name.
DT	{*name*} DT *value*,...	Allocates and initializes ten bytes of memory for each *value*. If the optional *name* is present, a variable of type TBYTE is created with the specified name.
ELSE	ELSE	Indicates the end of the true portion and the start of the false portion of a conditional block.
END	END {*entry*}	Indicates the end of the program module being assembled and optionally establishes the program's entry point as *entry*.

(cont.)

TABLE E-1, CONTINUED

Directive Name	Statement Syntax	Description
ENDIF	ENDIF	Signals the end of a conditional block.
ENDM	ENDM	Signals the end of a macro or repeat block.
ENDP	*name* ENDP	Signals the end of the procedure definition for *name*.
ENDS	*name* ENDS	Closes the segment named *name* or ends a structure definition for *name*.
EQU	*name* EQU *expression*	Assigns the value of *expression* to *name*. *Expression* can be an address expression, an assembler keyword, an arbitrary string of text, or a constant value.
.ERR	.ERR	Forces the generation of an assembler error.
.ERRB	.ERRB <*string*>	Forces the generation of an assembler error if *string* is blank.
.ERRDEF	.ERRDEF <*name*>	Forces generation of an assembler error if there is a symbol named *name* in the assembler's symbol table, and it was defined before the current source line.
.ERRDIF	.ERRDIF <*str1*>, <*str2*>	Forces generation of an assembler error if strings *str1* and *str2* are different.

(cont.)

TABLE E-1, CONTINUED

Directive Name	Statement Syntax	Description
.ERRE	.ERRE *expression*	Forces generation of an assembler error if given *expression* is false (evaluates to zero).
.ERRIDN	.ERRIDN <*str1*>, <*str2*>	Forces generation of an assembler error if strings *str1* and *str2* are identical.
.ERRNB	.ERRNB <*string*>	Forces the generation of an assembler error if *string* is not blank.
.ERRNDEF	.ERRNDEF <*name*>	Forces generation of an assembler error if there is not a symbol named *name* in the assembler's symbol table, or if *name* was defined after the current source line.
.ERRNZ	.ERRNZ *expression*	Forces generation of an assembler error if the given *expression* is true (evaluates to a non-zero value).
EVEN	EVEN	Aligns the location counter for the currently open segment to an even value by generating a NOP instruction (90H), if necessary.
EXITM	EXITM	Forces the current macro expansion or repeat block to terminate immediately.

(cont.)

TABLE E-1, CONTINUED

Directive Name	Statement Syntax	Description
EXTRN	EXTRN *name:type,...*	Signals to the assembler that the symbol named *name* is external to the current module and is of type *type*.
GROUP	*name* GROUP *segname,...*	Creates a definition for a group *name* with members from the list of one or more segments following the directive.
IF	IF *expression*	Assembles instructions in the true portion of a conditional block if *expression* is true (evaluates to a non-zero value).
IFB	IFB <*string*>	Assembles instructions in the true portion of a conditional block if *string* is blank.
IFDEF	IFDEF *name*	Assembles instructions in the true portion of a conditional block if there is a symbol named *name* in the assembler's symbol table, and it was defined before the current source line.
IFDIF	IFDIF <*str1*>, <*str2*>	Assembles instructions in the true portion of a conditional block if *str1* and *str2* are different.
IFE	IFE *expression*	Assembles instructions in the true portion of a conditional block if *expression* is false (evaluates to 0).

(cont.)

TABLE E-1, CONTINUED

Directive Name	Statement Syntax	Description
IFIDN	IFIDN <*str1*>, <*str2*>	Assembles instructions in the true portion of a conditional block if *str1* and *str2* are identical.
IFNB	IFNB *string*	Assembles instructions in the true portion of a conditional block if *string* is not blank.
IFNDEF	IFNDEF *name*	Assembles instructions in the true portion of a conditional block if there is no symbol named *name* in the assembler's symbol table, or if *name* was defined after the current source line.
INCLUDE	INCLUDE *filename*	Inserts the text contained in the source file specified by *filename* into the file currently being assembled.
IRP	IRP *fparam*, <*aparam*,...>	Start of repeat block that will be repeated once for each actual parameter *aparam*, with the actual parameter value being substituted for the formal parameter *fparam* in the text of the repeat block.

(cont.)

TABLE E-1, CONTINUED

Directive Name	Statement Syntax	Description
IRPC	IRPC *fparam, string*	Start of repeat block that will be repeated once for each character in *string,* with the current character being substituted for the formal parameter *fparam* in the text of the repeat block.
LABEL	*name* LABEL *type*	Creates a new variable or instruction label of type *type* with a value equal to the location counter for the current segment.
.LALL	.LALL	Causes all statements in macro expansions to be listed in the assembler listing file.
.LFCOND	.LFCOND	Causes text in false conditional blocks to be copied into the assembler listing file even though it is not being assembled.
.LIST	.LIST	Enables listing of program statements in the assembler listing file.
.LISTI	.LISTI	Enables listing of source lines from files included with the INCLUDE directive.

(cont.)

TABLE E-1, CONTINUED

Directive Name	Statement Syntax	Description
LOCAL	LOCAL *dname,...*	Creates one or more dummy names for use within a macro. The dummy name is replaced by an assembler-generated unique name of the form ??XXXX, where X is a hexadecimal digit when the macro is expanded.
MACRO	*name* MACRO *fparam,...*	Start of a macro definition block for macro *name*, with formal parameter names *fparam*.
NAME	NAME *modulename*	Sets the name of the module being assembled to the first 132 characters of *modulename*.
ORG	ORG *expression*	Sets the value of the location counter for the currently open segment to *expression*.
%OUT	%OUT *text*	Writes *text* to the user's terminal on both pass one and pass two of the assembly.
%OUT1	%OUT1 *text*	Writes *text* to the user's terminal on pass one of the assembly.
%OUT2	%OUT2 *text*	Writes *text* to the user's terminal on pass two of the assembly.

(cont.)

TABLE E-1, CONTINUED

Directive Name	Statement Syntax	Description
PAGE	PAGE {*length*}, {*width*}	Sets the page length and width for the listing file to *length* and *width*, respectively.
	PAGE +	Increments the section number used on the header line of listing file pages and generates a page break.
	PAGE	Generates a page break in the assembler listing file.
PROC	*name* PROC *type*	Initiates definition of a procedure called *name* of type *type*.
.PROT	.PROT	Enables assembly of protected instructions for the 80386 or 80286.
PUBLIC	PUBLIC *name*...	Makes the list of variables, instruction labels, or absolute symbols available to all other modules in the program.
PURGE	PURGE *macroname*	Has no effect.
.RADIX	.RADIX *expression*	Sets the default radix for numbers in the input file to *expression*. Numbers in *expression* are always evaluated in base 10, regardless of the current default radix.
RECORD	*name* RECORD *fname:width* {=*expr*},...	Defines a record called *name* with bit fields named *fname*. The field width, in bits, is given by the constant value *width* and the =*expr*, if present, defines the default value for the field.

(cont.)

TABLE E-1, CONTINUED

Directive Name	Statement Syntax	Description
REPT	REPT *expression*	Beginning of a block to be repeated *expression* number of times.
.SALL	.SALL	Suppresses the listing of macro expansions in the assembler listing file.
SEGMENT	*name* SEGMENT {*align*} {*combine*} {*useatr*} {*access*} {*class*}	Opens a program segment called *name* with attributes *align*, *combine*, *useatr*, *access*, and *class*.
.SFCOND	.SFCOND	Suppresses the listing of subsequent false conditional blocks in the assembler listing file.
STRUC	*name* STRUC	Signals the beginning of a structure definition.
SUBTTL	SUBTTL {*text*}	Sets the subtitle to be used on page headers in the listing file to *text*.
.TFCOND	.TFCOND	Toggles the current state of the "list false conditional blocks" flag.
TITLE	TITLE *text*	Sets the title to be used on page headers in the listing file to *text*.
.XALL	.XALL	Causes only statements in macro expansions which generate object code to be listed in the assembler listing file.
.XLIST	.XLIST	Suppresses listing of subsequent source lines in the assembler listing file.
.XLISTI	.XLISTI	Disables listing of source lines from files included with the INCLUDE directive.

80386 Register Names

TABLE F-1
80386 REGISTERS

Register Name	80386 Only	Description
EAX	yes	32 bit general register
EBX	yes	32 bit general register
ECX	yes	32 bit general register
EDX	yes	32 bit general register
ESI	yes	32 bit general register
EDI	yes	32 bit general register
EBP	yes	32 bit general register
ESP	yes	32 bit stack pointer
AX	no	16 bit general register; low word of EAX
BX	no	16 bit general register; low word of EBX
CX	no	16 bit general register; low word of ECX
DX	no	16 bit general register; low word of EDX
SI	no	16 bit index register; low word of ESI
DI	no	16 bit index register; low word of EDI
BP	no	16 bit base register; low word of EBP
SP	no	16 bit stack pointer; low word of ESP

(cont.)

TABLE F-1, CONTINUED

Register Name	80386 Only	Description
AH	no	high byte of AX
AL	no	low byte of AX
BH	no	high byte of BX
BL	no	low byte of BX
CH	no	high byte of CX
CL	no	low byte of CX
DH	no	high byte of DX
DL	no	low byte of DL
CS	no	code segment selector register
DS	no	data segment selector register
ES	no	extra data segment selector register
FS	yes	extra data segment selector register
GS	yes	extra data segment selector register
SS	no	stack segment selector register
CR0	yes	machine control register (includes 80286 machine status word)
CR2	yes	page fault linear address register
CR3	yes	page directory base register

(cont.)

TABLE F-1, CONTINUED

Register Name	80386 Only	Description
DR0	yes	linear breakpoint address 0 register
DR1	yes	linear breakpoint address 1 register
DR2	yes	linear breakpoint address 2 register
DR3	yes	linear breakpoint address 3 register
DR6	yes	breakpoint status register
DR7	yes	breakpoint control register
TR6	yes	test control register
TR7	yes	test status register
ST(0)-ST(7)	no	8087 floating point stack registers

Data Types and Ranges

386 | ASM supports the full range of 80386 and 80287 data types. Table G-1 lists the available data types and their value ranges.

All internal arithmetic done by 386 | ASM on integer values uses a 33 bit sign plus magnitude representation for integer values. This extends the possible value range for intermediate results during expression evaluation. The final result of expression evaluation is coerced to the appropriate data type, and an integer overflow error is reported if the result exceeds the allowable value range.

TABLE G–1
DATA TYPES AND RANGES

Data Type	80386 only	Directive	Bits	Description	Range (approximate)
BYTE	no	DB	8	80386 byte	-128 to 127 (signed integer) 0 to 255 (unsigned integer)
WORD	no	DW	16	80386 word 80287 word integer	-32,768 to 32,767 (signed integer) 0 to 65,535 (unsigned integer)
DWORD	no	DD	32	80386 double word 80287 short integer 80287 short real	-2^{31} to $2^{31}-1$ (signed integer) 0 to $2^{32}-1$ (unsigned integer) -3.4×10^{38} to -1.2×10^{-38}, 1.2×10^{-38} to 3.4×10^{38} } real
PWORD FWORD	yes	DP DF	48	80386 pointer word or a 16 bit segment selector plus a 32 bit unsigned offset	-2^{47} to $2^{47}-1$ (signed integer) 0 to $2^{48}-1$ (unsigned integer)
QWORD	no	DQ	64	80287 long integer 80287 long real	-2^{63} to $2^{63}-1$ (signed integer) 0 to $2^{64}-1$ (unsigned integer) -1.7×10^{308} to -2.3×10^{-308}, 2.3×10^{-308} to 1.7×10^{308} } real
TBYTE	no	DT	80	80287 packed decimal 80287 temporary real	$-10^{18}-1$ to $10^{18}-1$ (packed decimal) -1.1×10^{4932} to -3.4×10^{-4932}, 3.4×10^{-4932} to 1.1×10^{4932} } real

Expressions and Operators

An expression is a combination of constant and/or relocatable values (operands) and expression operators which evaluate to a single constant or relocatable value. Table H-1 lists the expression operators recognized by 386 | ASM.

Forward references to identifiers are permitted to occur in expressions provided the assumptions made by the assembler on pass one do not cause it to generate fewer bytes of object code on pass one than required. Common reasons for illegal forward references are forward references that would require a segment override byte to be generated, or to a label of type FAR. Illegal forward references can usually be corrected by judicious use of the PTR and segment override (:) operators. The fewer forward references there are in a source code file, the faster 386 | ASM assembles it.

The order of evaluation of expressions depends on operator precedence. Operators with highest precedence are evaluated first. Operators of equal precedence are evaluated from left to right. Table H-2 shows the operator precedence for the 386 | ASM expression evaluator. Operators on the same line in the table have equal precedence.

TABLE H-1
EXPRESSION OPERATOR SUMMARY

Operator	Relocatable Values Permitted	Syntax	Description
( )	yes	*(expr)*	Forces evaluation of the expression within parentheses before any adjacent operations.
[]	yes	*expr [basereg]* *expr [indexreg]* *expr1 [expr2]*	Used to generate based and/or indexed addressing modes. Also used as an index operator to add the values of two expressions.
.	yes	*expr:strfldname* *expr1:expr2*	The structure field name operator adds the offset of a field within a structure to the expression and sets the data type of the expression to the data type of the field. It is also used interchangebly with the + operator in order to add two expressions.
:	yes	*segreg:expr* *segname:expr* *groupname:expr*	The segment override operator forces the address of expr to be computed relative to the specified segment register, segment, or group. It also causes 386 l ASM to generate a segment override byte before the instruction, if necessary. A named segment or group must be assigned to a segment register with the ASSUME directive before it can be used with the segment override operator.
/	no	*expr1 / expr2*	Integer division with truncation of the fractional part.

(cont.)

TABLE H-1, CONTINUED

Operator	Relocatable Values Permitted	Syntax	Description
*	no	*expr1 * expr2*	Integer multiplication.
+	one operand only	*expr1 + expr2*	Addition.
-	yes	*expr1 - expr2*	Subtraction — both operands may be relocatable only if they are relative to the same segment, in which case the result is absolute. If one operand is relocatable, it must be the lefthand one.
+(unary)	yes	*+ expr*	Unary addition.
-(unary)	no	*- expr*	Unary subtraction.
.TYPE	yes	.TYPE *expr*	Returns a byte defining the mode and scope of expr.
AND	no	*expr1* AND *expr2*	Bitwise AND of the two operands.
EQ	no	*expr1* EQ *expr2*	Returns -1 if expressions are equal, zero if not equal.
GE	no	*expr1* GE *expr2*	Returns -1 if *expr1* is greater than or equal to *expr2*, zero otherwise.
GT	no	*expr1* GT *expr2*	Returns -1 if *expr1* is greater than *expr2*, zero otherwise.
HIGH	no	HIGH *expr*	Returns the high eight bits of a 16 bit value.
HIGHW	no	HIGHW *expr*	Returns the high 16 bits of a 32 bit value.

(cont.)

TABLE H-1, CONTINUED

Operator	Relocatable Values Permitted	Syntax	Description
LE	no	*expr1* LE *expr2*	Returns -1 if *expr1* is less than or equal to *expr2*, zero otherwise.
LENGTH	yes	LENGTH *varname*	Returns the number of elements in the variable, where a variable has more than one element by use of the DUP operator in the variable declaration statement, or by use of a list of initializers.
LOW	no	LOW *expr*	Returns the low eight bits of a 16 bit value.
LOWW	no	LOWW *expr*	Returns the low 16 bits of a 32 bit value.
LT	no	*expr1* LT *expr2*	Returns -1 if *expr1* is less than *expr2*, zero otherwise.
MASK	no	MASK *recnm* MASK *recfldnm*	Returns a bit mask for all the defined bits in a record, or for the bits in a record occupied by the specified field.
MOD	no	*expr1* MOD *expr2*	Modulo operator — returns the remainder after integer division.
NE	no	*expr1* NE *expr2*	Returns -1 if the expressions are not equal, zero if they are equal.
NOT	no	NOT *expr*	One's complement of the operand.
OFFSET	yes	OFFSET *expr*	Returns the number of bytes between the operand and the beginning of the segment in which it is defined. The returned value is relocatable.

(cont.)

TABLE H-1, CONTINUED

Operator	Relocatable Values Permitted	Syntax	Description
OR	yes	*expr1* OR *expr2*	Bitwise OR of the two operands.
PTR	yes	*type* PTR *expr*	Forces the expression to have the specified data type, where type is one of the set {BYTE, WORD, DWORD, PWORD, QWORD, TBYTE, NEAR FAR} or is an integer value which corresponds to the data type (please see the TYPE operator).
SEG	yes	SEG *expr*	Returns the segment selector for the segment to which *expr* is relative. The result is relocatable.
SHL	no	*expr1* SHL *expr2*	Shifts *expr1* to the left by the number of bits specified by the value of *expr2*. Bits shifted off the end of the expression are lost.
SHORT	yes	SHORT *label*	Forces the assembler to generate a one-byte offset in JMP instructions. Only necessary for forward references.
SHR	no	*expr1* SHR *expr2*	Shifts *expr1* to the right by the number of bits specified by the value of *expr2*. Bits shifted off the end of the expression are lost.
			(cont.)

TABLE H-1, CONTINUED

Operator	Relocatable Values Permitted	Syntax	Description
SIZE	yes	SIZE *varname*	Returns the number of bytes allocated for the specified variable. Equal to (LENGTH *varname*)*(TYPE *varname*).
THIS	no	THIS *type*	Creates an operand whose offset and segment values are equal to the current location counter value, and whose data type is given by one of the set {BYTE, WORD, DWORD, PWORD, QWORD, TBYTE, NEAR, FAR}.
TYPE	yes	TYPE *expr*	Returns a number representing the data type of *expr*. If the data type is a variable data type, the size in bytes of that data type is returned. If the data type is a label data type, -1 is returned for NEAR and -2 for FAR.
WIDTH	no	WIDTH *recordname* WIDTH *recfldname*	Returns the width, in bits, of specified record or record field.
XOR	no	*expr1* XOR *expr2*	Bitwise exclusive OR of the two operands.

TABLE H-2
OPERATOR PRECEDENCE

Operator	Precedence
LENGTH, SIZE, WIDTH, MASK, [ ]	highest
. (structure field name)	
: (segment override)	
OFFSET, SEG, TYPE, THIS	
HIGH, LOW, HIGHW, LOWW	
unary +, unary -	
PTR	
*, /, MOD, SHL, SHR	
+, -	
EQ, NE, LT, LE, GT, GE	
NOT	
AND	
OR, XOR	
SHORT, .TYPE	lowest

Symbol Types

User-defined symbols have a symbol type that is implicit in the way the symbol is defined. Table I-1 lists the symbol types supported by 386 | ASM.

TABLE I-1
SYMBOL TYPES

Symbol Type	Description	Directive(s) Used to Create Symbol
group	Name of a group used to group one or more segments.	GROUP
segment	Name of a segment. Carries attributes of align type, combine type, protection, etc.	SEGMENT
variable	Address of data within a particular segment. Carries a data type of BYTE, WORD, DWORD, PWORD, QWORD, or TWORD.	DB, DW, DD, DQ, DP, DF, DT, LABEL, EQU, =
structure	Address of data area allocated by invoking a structure definition template.	name of structure definition
record	Address of data area allocated by invoking a record definition template.	name of record definition
label	Address within a segment used for control transfer by JMP or CALL instructions. Carries a data type of NEAR or FAR.	LABEL, =, EQU
procedure label	Exactly the same as *label*.	PROC

(cont.)

TABLE I-1, CONTINUED

Symbol Type	Description	Directive(s) Used to Create Symbol
constant	A constant integer value.	EQU, =
alias	Another name for an assembler reserved word.	EQU
text substitution	A string of ASCII text that is substituted for the symbol name wherever it is used.	EQU
macro	A name of a macro definition which, when invoked, causes the macro to be expanded.	MACRO
structure definition	A name of a structure definition template which, when invoked, causes allocation and initialization of a data area whose size and initial values are defined in the template.	STRUC
structure field	A name of a field within a structure template. Carries a data type (BYTE, WORD, DWORD, PWORD, QWORD. TWORD), the offset it is located at within the structure template, and an initial value.	DB, DW, DD, DQ, DP, DF DT within a structure definition block
record definition	A name of a record definition template which, when invoked, causes allocation and initialization of an 8 bit, 16 bit or 32 bit data area.	RECORD
record field	A name of a bit field within a record template. Carries the offset (in bits) where it is located in the record template, a width (in bits), and an initial value.	RECORD

Mixing USE16 and USE32 Segments

Instructions in USE16 segments use 16 bit operands and addressing modes by default. Instructions in USE32 segments use 32 bit operands and addressing modes by default. However, the alternate form of either the operand size or the addressing mode can be selected independently by prefixing the instruction with either (or both) an operand size override byte or an address size override byte. 386 | ASM automatically generates these override bytes as necessary. This appendix discusses some of the issues that arise in the case of the 16 bit and 32 bit forms of certain instructions, and in the interactions between code in USE16 and USE32 segments.

J.1 Intersegment Procedure Calls

When a FAR procedure call is made between segments with different use types, 386 | ASM generates the form of the CALL instruction which corresponds to the use type of the destination segment. This means that the default form of the RET instruction can always be used, regardless of whether the procedure was called from a segment with a different use type.

Example:

```
cseg1   SEGMENT  use16
p1      PROC     far
        ret                      ; 16 bit far return

p1      ENDP
cseg1   ENDS

cseg2   SEGMENT  use32
        call     p1              ; assembler automatically
                                 ; generates 16 bit form
```

```
                                        ; of far call instruction
cseg2   ENDS

cseg3   SEGMENT   use16
        call      p1            ; 16 bit form of call
cseg3   ENDS
```

This works nicely on FAR calls from a USE16 segment to a USE32 segment. The CALL pushes the full 32 bits of EIP onto the stack, and the RET (in the USE32 segment) restores it. Note, however, that there is a potential problem with FAR calls from a USE32 segment to a USE16 segment. Since the 16 bit form of CALL and RET are used, only IP (not EIP) is saved on the stack and restored. Thus, if any code changes the high 16 bits of EIP between the CALL and the RET, the RET will not restore control to the correct location! Normally this is not a problem since, when executing in a USE16 segment, there is no reason to modify the high 16 bits of EIP. There is a workaround if it is possible that the high 16 bits of EIP might be modified. 386 | ASM can be forced to generate the 32 bit form of the CALL by using an indirect call (please see J.2). The 32 bit form of the RET can be forced by hardcoding an operand size override byte before the RET instruction. Note that if this is done, the procedure can no longer be called with a 16 bit form of the CALL instruction!

Example:

```
cseg1   SEGMENT   use16
p1      PROC      far
        DB        66h           ; operand size override
        ret                     ; gives 32 bit form
                                ; of return
p1      ENDP
cseg1   ENDS

dseg    SEGMENT
p1_call LABEL     pword         ; indirect 32 bit far call
        DD        OFFSET p1
        DW        cseg1

dseg    ENDS

cseg2   SEGMENT   USE32
        call      p1_call       ; force 32 bit form
                                ; of call
```

```
cseg2   ENDS

cseg3   SEGMENT   USE16
        call      p1          ; this won't work
                              ; because of 32 bit
                              ; return in proc p1
cseg3   ENDS
```

J.2 Indirect Control Transfer

Indirect calls and jumps in protected mode pose an ambiguity problem for the assembler because of the possibility of mixing USE16 and USE32 segments in the same program. When making an indirect call/jump from a USE16 segment, the data type of the variable being used with the indirect call/jump can be used to decide what form of the instruction to generate.

Variable Data Type	Call/Jump
WORD	NEAR call/jump to USE16 segment
DWORD	FAR call/jump to USE16 segment
PWORD/FWORD	FAR call/jump to USE32 segment

When making an indirect call/jump from a USE32 segment, however, the data type is ambiguous.

Variable Data Type	Call/Jump
DWORD	FAR call/jump to USE16 segment
DWORD	NEAR call/jump to USE32 segment
PWORD/FWORD	FAR call/jump to USE32 segment

386|ASM, therefore, uses the convention that an indirect call/jump through a variable of type WORD when executing in a USE32 segment means a far call/jump to a USE16 segment. Table J–1 summarizes how data types determine the type of instruction generated for an indirect call/jump. The following example generates all three kinds of indirect control transfers from a USE32 segment.

Example:

```
data    SEGMENT
far32   DP        ?                  ; seg:offset in USE32 seg.
near32  DD        ?                  ; offset in USE32 seg.
far16   DD        ?                  ; seg:offset in USE16 seg.
data    ENDS

code    SEGMENT   use32
        call      far32              ; FAR call to USE32 seg.
        jmp       near32             ; NEAR jump within seg.
        call      word PTR far16     ; force FAR call to
                                     ; USE16 seg.
code    ENDS
```

TABLE J-1
INDIRECT CONTROL TRANSFER DATA TYPES

Use Type of Segment Call is Made From	Variable Data Type	Type of Control Transfer Generated
	WORD	NEAR call/jump within USE16 segment
	DWORD	FAR call/jump to another USE16 segment
USE16	PWORD/FWORD	FAR call/jump to a USE32 segment
	16 bit register	NEAR call/jump within USE16 segment
	32 bit register	illegal
	WORD	FAR call/jump to a USE16 segment
	DWORD	NEAR call/jump within USE32 segment
USE32	PWORD/FWORD	FAR call/jump to another USE32 segment
	16 bit register	illegal
	32 bit register	NEAR call/jump within USE32 segment

J.3 Data Width for Stack PUSH/POP

The 80386 processor pushes either WORD (16 bit) or DWORD (32 bit) values on the stack. The default operand size for PUSH/POP (as for all instructions) is WORD when executing in a USE16 segment, DWORD for a USE32 segment. If the operand to the PUSH/POP instruction does not have a data type, 386 | ASM will not generate an operand size override byte, so the default operand size will be pushed/popped. The PTR operator can be used to give the operand a data type of DWORD or WORD to force the assembler to use the desired operand size.

Example:

```
cseg    SEGMENT     use32

        push        0               ; push a DWORD
        push        word PTR 0      ; push a WORD
        push        dword PTR 0     ; push a DWORD

cseg    ENDS
```

When segment registers are pushed and popped, the 80386 pushes/pops 16 bits when executing in a USE16 segment, and 32 bits (of which the high 16 bits are unused and pushed as zero) in a USE32 segment. It is permissible to use an operand size override byte to change the default operand size, but 386 | ASM does not provide any automatic way to generate an override byte. If an override byte is desired, it must be coded with a DB.

Example:

```
cseg16    SEGMENT   use16
          push      ds          ; push 16 bits
          pop       ax
cseg      ENDS
cseg32    SEGMENT   use32
          push      eax
          pop       ds          ; pop 32 bits
          DB        66h         ; operand size override
          push      ds          ; push 16 bits
cseg32    ENDS
```

The PUSHF/POPF (push/pop flags) instructions also have two forms: the 16 bit form pushes/pops the low 16 bits of the EFLAGS register; and the 32 bit form pushes/pops the full 32 bits of the EFLAGS register. Table J–2 shows what instruction mnemonics are used to generate the desired instruction form. The programmer explicitly specifies which form of the instruction is to be generated by his choice of instruction mnemonics.

TABLE J-2
PUSH/POP FLAGS INSTRUCTIONS

Mnemonic	Instruction Generated
PUSHF	Push low 16 bits of EFLAGS
POPF	Pop WORD into low 16 bits of EFLAGS
PUSHFD	Push all 32 bits of EFLAGS
POPFD	Pop DWORD into EFLAGS

Example:

```
cseg16      SEGMENT  use16
            pushf              ; push low 16 bits of EFLAGS
            popfd              ; pop DWORD into EFLAGS
cseg16      ENDS

cseg32      SEGMENT  use32
            pushfd             ; push all 32 bits of EFLAGS
            popf               ; pop WORD into low 16 bits of
                               ; EFLAGS
cseg 32     ENDS
```

J.4 Interrupt Return Instruction

The interrupt return (IRET) instruction has a 32 bit and a 16 bit form. Protected mode interrupt handlers should always use the 32 bit form, and real mode interrupt handlers should always use the 16 bit form. 386 | ASM generates the 32 bit form of the instruction when the IRETD mnemonic is

used, and generates the 16 bit form of the instruction when the IRET mnemonic is used.

Example:

```
cseg    SEGMENT    use16

        iret                    ; interrupt return for
                                ; real mode handler

        iretd                   ; interrupt return for
                                ; protected mode handler

cseg    ENDS
```

J.5 Load/Store Descriptor Table Registers

The load and store instructions for the GDTR and IDTR registers have a 16 bit and 32 bit form. The 16 bit form is provided for compatibility with the 80286 and only loads/stores 24 bits of the descriptor table base address. The 32 bit form loads/stores a full 32 bit base address. 386|ASM always generates the 32 bit form when the suffix "E" is used with the instruction mnemonic, and generates the 24 bit form when it is not used. Table J–3 shows which instruction mnemonics generate which form of the instruction.

TABLE J–3
DESCRIPTOR TABLE REGISTER INSTRUCTIONS

Mnemonics	Instruction Generated
LGDT/SGDT	16 bit (80286) form of load/store GDTR
LGDTE/SGDTE	32 bit (80386) form of load/store GDTR
LIDT/SIDT	16 bit (80286) form of load/store IDTR
LIDTE/SIDTE	32 bit (80386) form of load/store IDTR

Appendix K

Easy OMF-386 Object File Format

Easy OMF-386 is the format used by 386 | ASM for object files generated for the 80386 (object files generated for the 8086, 8088, 80186, and 80286 are in OMF-86 format). Easy OMF-386 is a simple extension of the OMF-86 used by Intel and Microsoft. This appendix describes the differences between Easy OMF-386 and OMF-86. For a description of OMF-86, please see references 1 and 2.

386 | ASM signals to the linker that an object module is targeted for the 80386 by placing the following comment record at the beginning of an object module:

0	1	2	3 ... 7	8
88H	80H	AAH	'80386'	
Record Type	Flags	Class		Checksum

where "H" designates a hexadecimal value.
The 80386 comment record should be located immediately after the module header record (THEADR) and before any other records of the object module.

The other records of an object module are formatted in Easy OMF-386 the same way as in OMF-86, except that any offset, displacement, or segment length field of an object record is four bytes long instead of two bytes. The following records contain the fields which increase in size:

Record:	Field:
SEGDEF	Offset and segment length
PUBDEF	Offset
LEDATA	Offset

Record:	**Field:**
`LIDATA`	Offset
`Explicit FIXUPP`	Target displacement
`BLKDEF`	Return address offset
`LINNUM`	Offset
`MODEND`	Target displacement
`DEBSYM`	Offset
`LOCSYM`	Offset

In FIXUPP records, the following two new "Loc" values have been defined:

5	32 bit offset
6	Base + 32 bit offset (long pointer)

In SEGDEF records, the following two new "Align" values have been defined:

5	Relocatable and doubleword-aligned segment (segment begins on a doubleword boundary)
6	Relocatable and 4K page-aligned segment (segment begins at an address on a 4096-byte boundary)

Example 386 | ASM Command Lines

1. Assemble the file "HELLO.ASM" to produce an object file "HELLO.OBJ" and a listing file "HELLO.LST":

IBM PC/MS-DOS:

```
386ASM HELLO
```

VAX/VMS or UNIX:

```
XA386 HELLO
```

2. Assemble the file "HELLO.ASM" and discard the listing file:

IBM PC/MS-DOS:

```
386ASM HELLO -NOLIST
```

VAX/VMS or UNIX:

```
XA386 HELLO -NOLIST
```

3. Assemble the file "HELLO.ASM" and discard the object file:

IBM PC/MS-DOS:

```
386ASM HELLO -NOOBJECT
```

VAX/VMS or UNIX:

```
XA386 HELLO -NOOBJECT
```

4. Assemble the file "HELLO.ASM" for the 8086:

IBM PC/MS-DOS:

```
386ASM HELLO -8086
```

VAX/VMS or UNIX:

```
XA386 HELLO -8086
```

5. Assemble the file "HELLO.ASM" and redirect the listing and object files to another directory:

IBM PC/MS-DOS:

```
386ASM HELLO -LIST \LISTINGS\HELLO -OBJECT \OBJS\HELLO
```

VAX/VMS:

```
XA386 HELLO -LIST [LISTINGS]HELLO -OBJECT [OBJS]HELLO
```

UNIX:

```
X386ASM HELLO -LIST /LISTINGS/HELLO -OBJECT /OBJS/HELLO
```

6. Assemble the file "HELLO.ASM" and write all errors to the file "HELLO.ERR":

IBM PC/MS-DOS:

```
386ASM HELLO -ERRORLIST HELLO
```

VAX/VMS or UNIX:

```
XA386 HELLO -ERRORLIST HELLO
```

Index

A

B

C

D

E

F

G

H

I

M

N

O

P

T

U

V

W

X

The people who wrote, edited, revised, reviewed, indexed, formatted, polished, and printed this manual were:

John Benfatto, Alan Convis, Noel Doherty, Lorraine Doyle, Bryant Durrell, Nan Fritz, Elliot Linzer, John Mann, Bob Moote, Kim Norgren, Cate O'Hara, Amy Weiss and Rick Wesson.

60 Aberdeen Avenue, Cambridge, MA 02138
617-661-1510 Fax: 617-876-2972